Master's Thesis

M.Sc. Majority Machine Intelligence

Increasing Horizon Policy Neural Networks for Finite-Horizon MDP's

submitted by:

David Sutter

Matriculation No.:

19-066-919

E-Mail:

david.sutter@stud.unibas.ch

Examiner:

Prof. Dr. Malte Helmert

Supervisor:

Dr. Thomas Keller

Submission date, place:

06.04.2022, Basel

Acknowledgement

First of all, I would like to express my gratitude to Prof. Dr. Malte Helmert, who gave me the opportunity to write my thesis on this topic in the Artificial Intelligence research group. Furthermore, I would like to thank Dr. Thomas Keller, who has supported me during the preparation phase. Both the encouragement to work out further methods independently and the contribution of his thoughts have led to a valuable discourse for this work. In particular, the critical questioning of my approaches showed me inaccuracies and possibilities for improvement, which had a great impact to the elaboration.

Abstract

To solve stochastic state-space tasks, the research field of artificial intelligence is mainly used. PROST2014 is state of the art when determining good actions in an MDP environment. In this thesis, we aimed to provide a heuristic by using neural networks to outperform the dominating planning system PROST2014. For this purpose, we introduced two variants of neural networks that allow to estimate the respective Q-value for a pair of state and action. Since we envisaged the learning method of supervised learning, in addition to the architecture as well as the components of the neural networks, the generation of training data was also one of the main tasks. To determine the most suitable network parameters, we performed a sequential parameter search, from which we expected a local optimum of the model settings. In the end, the PROST2014 planning system could not be surpassed in the total rating evaluation. Nevertheless, in individual domains, we could establish increased final scores on the side of the neural networks. The result shows the potential of this approach and points to eventual adaptations in future work pursuing this procedure furthermore.

Erklärung zur wissenschaftlichen Redlichkeit

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)

Bachelor's / Master's Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:
INCREASING HORIZON POLICY NEURAL NETWORKS FOR FINITE-HORIZON MDP'S

First Name, Surname *(Please print in capital letters)*: DAVID, SUTTER

Matriculation No.: 19-066-919

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

■ Yes ☐ No

Place, Date: Schopfheim, 06.04.2022

Signature: D. Sutter

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .

UNI
BASEL

Contents

List of Figures

List of Tables

1 Introduction

In everyday life, we are constantly confronted with tasks of the most diverse kinds that need to be solved. While simple jobs allow rather low scope for interpretation or variation, more complicated circumstances require prior considerations. Successful planning of meaningful behavior enables, in addition to the solution finding itself, the distinction of the respective quality. In this way, limited resources such as spent execution time can be reduced.

Since planning plays an important role in a variety of environments, it is hardly surprising that it is a current area of research. The simplest variant is called *classical planning*, which refers to deterministic and fully observable environments. Thus, the current state is completely declared and every action leads to a known successor state. The goal of planning is on the one hand the verification whether a given task is solvable and on the other hand the determination of a plan, in form of a sequence of actions that leads to the target state, as efficiently as possible. One exemplary use case is the *15-puzzle* as seen in Figure 1. Here the actions consist of moving a tile to an adjacent free space and the aim is to sort the numbers in ascending order. By knowing which action leads to a certain state, a path can be established due to blind search algorithms, at least in small state-spaces, i.e. with comparatively few possible states.

However, such deterministic and fully observable settings are rarely the case outside of such a clearly defined game. In real world scenarios, one often encounters uncertainty in the determination of a successor state given an action. Thus, it can happen that despite the expectation of a result as a consequence of the chosen action another state is reached. In case of the 15-puzzle, Figure 2 illustrates the related probabilistic version. This kind of uncertainty can be modeled with stochastic planning. While we have searched for a path in classical planning, this is not possible for stochastic planning due to the probabilistic component, because it cannot be guaranteed that an action that led to the goal in one run will do so the next walkthrough. Therefore, we need to change our approach and instead look for a policy that provides the most promising action in a given state. Although the probabilistic approach may seem needlessly complicated, it is worth taking a closer look, because there is a large area of stochastic game and real world tasks. It is often quite easy to identify multiple solutions, but they can vary greatly in reliability. By using current computer technology, simulations of individual actions can be performed quickly, but large systems usually remain intractable.

Another state-of-the art approach in artificial intelligence is that of machine learning. This technology includes the topic of neural networks, which are inspired on the biological neu-

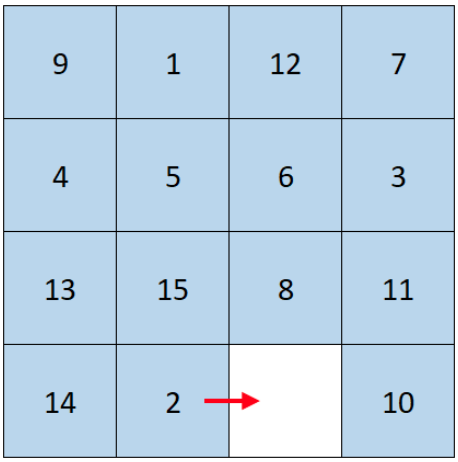

Figure 1: Deterministic and fully observable environment as 15-puzzle, moving number *2* to the right

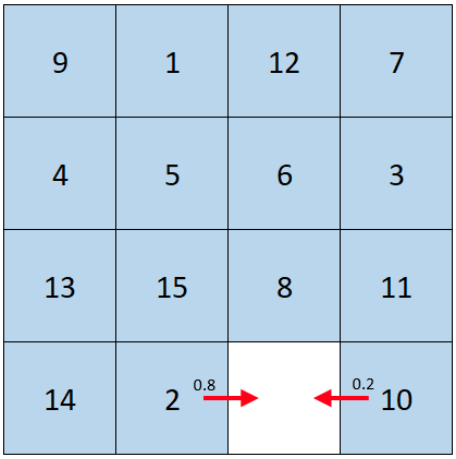

Figure 2: Stochastic and fully observable environment, action of moving right has several possible outcomes

rons in a brain. If many of these neurons are connected, a neural network is created which enables predictions to be made about future events based on training data. This approach is very different from the previous one, since the purpose of planning is to determine the impact before an actual execution of action. Even if both procedures seem contradictory, there are systems that take advantage of both of them. This thesis aims to build such an associated hybrid-application of planning task and neural network. Thus, a policy should be developed in a stochastic environment that mitigates the disadvantages of both approaches and thus outperforms one-sided methods. There are many similar elaborations on this topic, such as "Human-level control through deep reinforcement learning" (Minh et al 2015). The authors describe the application of reinforcement learning method in order to create a deep Q-network that learns policies from high-dimensional inputs fed to the model. In our case, however, we will use the supervised learning method, i.e. we will generate the trained data by our own, which will then be used to train the model. In the next section, the basics will be explained in more detail in order to introduce the intentions of the two variants of solutions approaches. In addition, the stochastic planner PROST is described as it serves as a foundation for implementations in the following chapters. In the third section, we present a detailed description of the intention of this work. Chapter 4 defines the setup of the experiments, which are explained in the following section. Upcoming is the evaluation of the obtained results, as well as an insight into further adjustments to optimize the outcomes. In the final section we share our assessment about the work and give an outlook of potential future work.

2 Background

In order to introduce the following topics in a comprehensible way, first we focus on the fundamentals. For this reason, we start with a formal description of the system properties in which probabilistic planning is applied.

2.1 Markov Decision Process

A Markov Decision Process (MDP) denotes a state-space with uncertainty regarding the transition between actions and their associated states. Contrary to deterministic environments, the same action can result in different successor states. An MDP contains the name Markov due to the Markov property, which specifies that the probability and reward distribution only depend on the current state and action, but not on previous states and actions. In formal terms, an MDP is represented by a 6-tuple $\mathcal{T} = \langle S, A, P, R, s_0, H \rangle$ (Puterman 1994; Russel and Norvig 2010). Decisions are spread over different time steps, so called *decision epochs*. H determines the number of decision epochs or steps, which restricts the domain to a so called finite-horizon MDP. An infinite-horizon MDP would allow infinite sequences of decisions in terms of actions, but in this thesis we will refer to finite-horizon MDP's. S denotes a finite set of states, where s_0 designates the initial state and A is a finite set of actions. P declares the *transition probability function* where $\sum_{s' \in S} P(s'|s,a) = 1$ and $P(s'|s,a) \in [0,1]$ determines the probability of ending up in $s' \in S$ when applying action $a \in A$ in $s \in S$. Finally the *reward function* $R(s,a)$ maps a value to a state $s \in S$, at a certain decision epoch.

Since MDP's are growing rapidly to intractable size, there is a more compact description called *factored* MDP(Boutilier, Dearden, and Goldszmidt 2000). Such a factored representation becomes exponentially less large-scaled than the explicit representation by describing the environment with state variables. The adjustment consists of representing states S in terms of valuation of state variables V. Similarly, P can be described as changes of state variables and R as a function over them, which maps to some real-value as reward. As mentioned in the previous section, the intention of MDP's is not to achieve a *path* , but to compute a *policy* π, which maps to each state s an action a, that maximizes the expected accumulated reward the action would lead to. In order to determine the expected reward of action a in state s, we have to take all possible successor states $s' \in S$ into account. Considering the applied action in a state as a random variable X with a finite number n of outcomes $s'_1, ..., s'_n$ and probability of occurrence $P(s'_i|s,a)$, for $1 \leq i \leq n$, we can determine

the reward for choosing a certain action a. In order to evaluate the policy π itself in the current state s we can apply the *state-value function*, considering current depth d, (Bellman 1957; Sutton and Barto 1998) where $V_\pi : S \to \mathbb{R}$:

$$V_\pi(s,d) = \begin{cases} R(s,\pi(a)) + \sum_{s' \in S} P(s'|s,\pi(a)) \cdot V_\pi(s',d)), & \text{if } d > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

If this cumulative approach is applied over all possible states and their successors, the determination of the maximum value leads to the optimal policy. Since the applied action depends on the current policy π as well as the state-value of potential successor states, the *action-value function* Q_π, with $Q_\pi : S \times A \to \mathbb{R}$ can be derived as:

$$Q_\pi(s,d,a) = R(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot Q_\pi(s',d,\pi(s')) \tag{2.2}$$

The maximal expected reward in a certain state s can be achieved by choosing the action which maximizes the expected reward of the action-value Q_*. Formally, the *Bellman Equation* describes this setting as follows:

$$V_*(s,d) = \begin{cases} \max\limits_{a \in A} Q_*(s,d,a), & \text{if } d > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

$$Q_*(s,d,a) = R(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot V_*(s',d) \tag{2.4}$$

V_* is the maximal expected reward we can achieve from state s. Q_* is determined by adding the current action-value $R(s,a)$ and the sum over the maximal expected reward of all possible successor $s' \in S$ times the transition probability. To obtain a value for this function, all successors passed and their maximal expected reward determined. In general, it is intractable to calculate all maximal expected rewards over the entire state-space of an MDP. An optimal policy is the policy greedy with respect to the Bellman equation.

2.2 Neural Networks

Learning pursues a completely different approach compared to planning. The difference is already noticeable when looking at the initialization of the task. As shown above in the domain of MDP's, planning procedures act in environments with background knowledge about the set of rules governing the system, i.e. definitions regarding states, actions or rewards, which determine the behaviour when acting. Learning leaves it to an agent itself

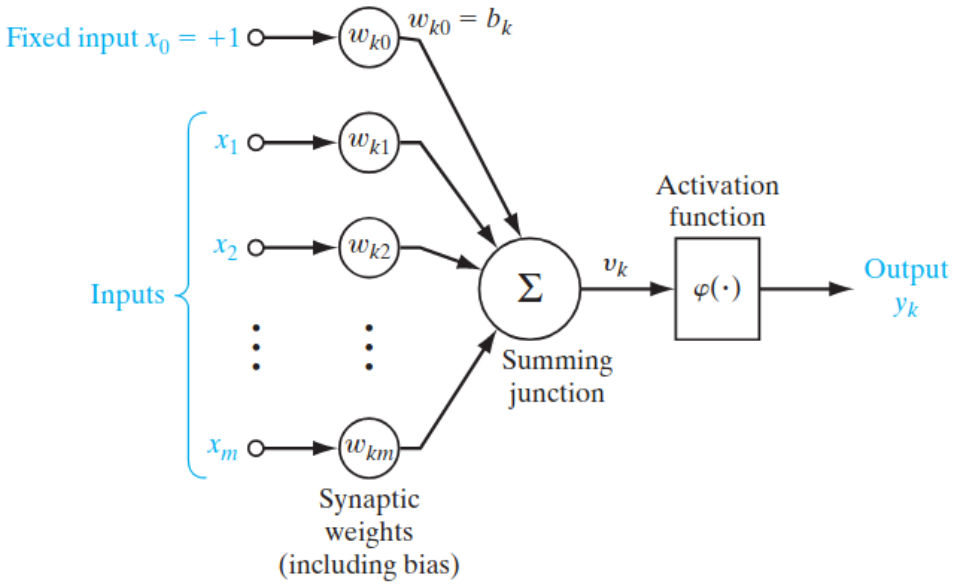

Figure 3: General model of an artificial neuron (from Haykin 2009)

to interact in the environment without this kind of information. By doing so, the results are evaluated and the outcome passed to the learner, which influences the behavior in the following trial. Based on collected experiences over many shots, the agent learns to interact with the environment on its own, without even knowing what it actually looks like.

Machine learning (Russel and Norvig 2010) is state of the art when aiming to improve the performance of a system in a certain environment by making observations, like using labeled training samples. Labeled training sets are tuples of correlated elements in terms of an input and the desired result. Such a data structure is required to prepare a *supervised learning* method with the intention to make own predictions on future data. A commonly used learning architecture is represented by the neural networks (NN). Neurons are processing units, which are preparing some input and produce a certain outcome and form, when interconnected, the basis of an NN (Haykin 2009). Figure 3 illustrates the main components of a neuron. The *input signal* consists of a vector $x^t = (x_1, ...x_m)$, where each $x_i \in \mathbb{R}$, with the information provided to the neuron. This might encode color values of pixels in an image or, in the case of MDP's, the current values of the state variables V. Each input is weighted by its own value w that influences the degree of impact a certain information has. The bias b_k is a constant which is always 1 and has a weight just like the other inputs. Taking this additional value into account enables transposing the resulting output by constant size. In a certain neuron k the weighted sum of the input and bias parameter is passed as input v_k

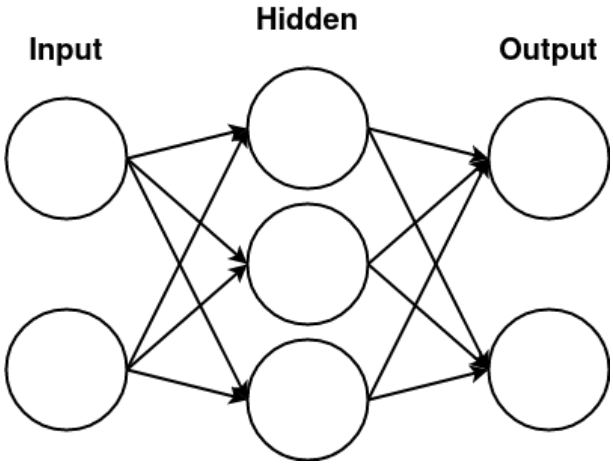

Figure 4: Abstract model of a fully connected neural network

to an activation function, where

$$v_k = b_k + \sum_{i=1}^{m} w_{ki} x_i \tag{2.5}$$

The final output of the neuron is then defined by

$$y_k = \varphi(v_k) \tag{2.6}$$

An NN is an interconnected directed graph of several nodes, where each node represents a neuron. The nodes are arranged in so called layers and the outcome serves as weighted input of the following layer. The first layer is called input-layer and the last one output-layer.

The functionality that makes the NN a powerful architecture is the learning approach. For this purpose the weight of the input, as well as the one from the bias, parameters are adjusted, depending on the result and its inaccuracy. The errors are measured by comparing the computed outcome and the provided correct labels of the training set. Often a subset of the training data is retained as test set to evaluate the accuracy of the NN after training.

2.3 Model Fitting

In order to achieve an increasingly accurate approximation to a target value, a procedure is needed to improve the parameters of an NN. A frequently used model optimization algorithm is gradient descent (Murphy 2012). The objective of such an optimization involves minimizing the difference of the target value y and the prediction produced of the network

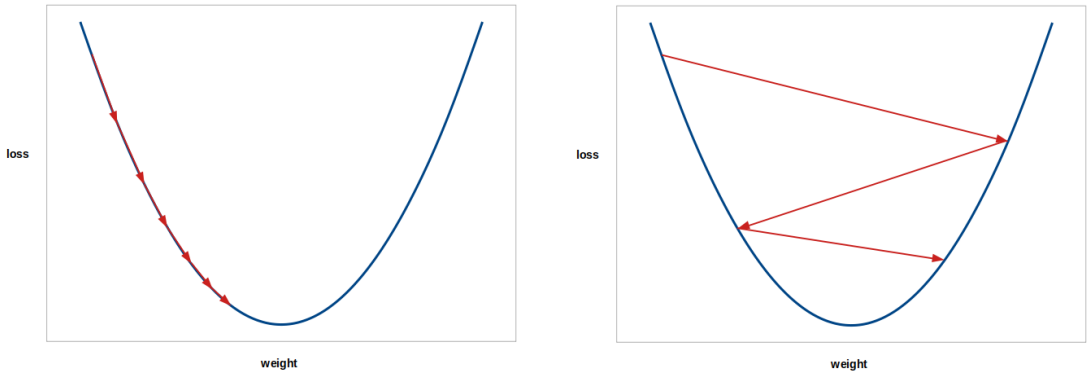

Figure 5: Gradient descent optimization using low step size left and large step size right

$\hat{y}$, i.e. minimizing the loss function. There are several types of loss functions. The most basic variant merely calculates the difference between two numerical values.

$$J = |\hat{y} - y| \tag{2.7}$$

To minimize the loss the parameters θ of the model have to be optimized, which in the case of the NN are the respective weights. Since loss functions determine the behavior for values deviating from the target function they have a big impact on learning behavior. As illustrated in Figure 5 different values of the weight ends up in another loss result. Gradient descent enables adjusting the weight in direction of the local steepest descent. In order to do so, it takes the gradient of the loss function J in respect to the weights θ for each prediction and target i of n samples.

$$\nabla J(\theta) = \sum_{i=1}^{n} (\hat{y}_i - y_i) \tag{2.8}$$

The model fitting procedure introduces the model weights θ at the current time step t, where the learning rate α defines the step size. Thus, the optimization algorithm can be represented as follows:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t) \tag{2.9}$$

Here θ_t is updated for the steepest descent, given by the gradient of the loss function. Since the updating range depends on the gradient of the loss function, the steps in Figure 5 decreases when approaching the minimum, because the loss itself decreases. The specification of the learning is an elementary part of the optimization procedure. Choosing the initial value too high causes too coarse steps, which leads the optimization to comparatively

poor results. Conversely, a low learning rate means that a large number of optimization steps have to be applied, which is why the training process takes significantly more time. In addition, a small step size may get stuck in a local minimum and never reach the global optimum. An ideal value cannot be determined in a generalized way and depends on the respective application and the network.

2.4 Probabilistic Planning PROST

In the first section we mentioned the probabilistic planning system PROST (Keller and Eyerich, 2012). Since this thesis will be based on the framework provided by PROST, it is necessary to make its operating principle clear. First we introduce the PROST version of 2011, followed by the latest version of 2014 in the next subsection. PROST already participated with success in the International Probabilistic Planning competition (IPPC) and is thus a promising starting point in handling MDP's.

2.4.1 PROST - 2011

First, it requires some form of definition of the actual task. For this the Relational Dynamic Influence Diagram Language (RDDL) (Sanner 2010) is a fitting description of stochastic domains. RDDL enables a compact description of MDP's, where states and actions are defined by variables. Into this manner states can be defined in terms of digit sequences, where each digit stands for domain dependent conditions, such as *is-open* in the *elevators* domain. The corresponding action behaves in a similar manner, thus these are displayed as a sequence as well and determine which actions are applied. This approach enables the use of several actions at once, where the planner perceives them as only one. In this way the elevators domain example enables both, moving elevator 1 up and moving elevator 2 down at the same time.

The *Upper Confidence Bounds applied to Trees* (UCT) algorithm (Kocsis and Szepesvári 2006) provides the basic framework for the probabilistic planning system. UCT belongs to the family of Monte Carlo Tree Search (MCTS) algorithms (Browne et al. 2012), which aim to approximate optimal decisions in large state-space MDP's. In order to determine a solution, MCTS focuses on expanding a search tree. The algorithm consists of the four steps *selection, expansion, simulation* and *backpropagation*. The first step *selection*, starts at the root of the search tree and walks through it by selecting child nodes until a leaf node is reached. If the reached leaf is not a terminal state a new node is *expanded*. Next the *simulation* is applied, which designates the *rollouts*, consisting of random decision until a

terminal state is reached. At last the result achieved from the simulation is *backpropagated* to the related parents until the root and updates the information of each passed node.
Keller and Eyerich introduce two types of nodes in the search tree, so called *chance nodes* and *decision nodes*. In a decision node a decision is made which action will be executed. The edge of the respective action leads to a chance node. From this node further edges go off, which are associated with a probability value. The choice of the next connection depends in chance nodes on the randomness according to the assigned probability values. The successor is again a decision node and so on. When passing a decision node in the rollouts, UCT comes to a decision whether to choose an action, due to the promised rewards or when a node was not frequently selected in previous rollouts. However, unvisited childs of decision nodes are priored and selected randomly first. In the other case UCT exploits the node maximizing the term:

$$B\sqrt{\frac{log\ n.N^k}{n_i.N^k}} + n_i.R^k \tag{2.10}$$

Here $n.N^k$ is defined as the total number of visits in the current node n and $n_i.N^k$ the total number of visits in the respective successor node. This formula ensures that even with an outstanding expected reward in a successor node n_i, with sufficiently growing number of rollouts N^k among n_i at a certain point in time the other nodes are visited as well. B denotes a bias, which balance both *exploration* and *exploitation*. In case of a chance node UCT performs trials of the possible actions with their probability of different outcomes $P(s'|s,a)$ in the current node and evaluate their result.

A particularly important efficient step to achieve good results in the decision tree search depicts the limitation of the depth search. Since UCT needs intractable many rollouts to converge to an optimal policy, Keller and Eyerich improve the process through a profitable method that consists of relaxing the factored MDP $T = \langle V, A, P, R, H, s_0 \rangle$ to $T = \langle V, A, P, R, H^\lambda, s_0 \rangle$. By limiting the search depth, the duration of searching is decreased, where no optimal policy for T can be found. However, a policy for T^λ can be determined even for large state-spaces, which according to Keller's and Eyerich's experiments often overcome trials in finding an optimal policy for T. Due to the stochastic behavior in MDP's increasing number of applied actions cause an even greater uncertainty. Thus, far-sighted decisions have a diminishing influence on the reward of the current state. In order to reduce the effort during exploration the set of actions is restricted to a reasonable subset. Therefore Keller and Eyerich introduce the definitions for dominating and equalized action pairs.

Definition 1 *Action a dominates action a' in state s (written $a >_s a'$) if $P(s'|a,s) = P(s'|a',s)$ and $R(s,a) \geq R(s,a')$*

Definition 2 *Two actions a and a' are equivalent in state s (written $a =_s a'$) if $P(s'|a,s) = P(s'|a',s)$ and $R(s,a) = R(s,a')$*

In this way, redundant actions can be removed from the MDP, and therefore exploration focused on more promising decision nodes. Since there occur many of such dominating or equalizing correlations, this approach has an enormous impact on size of the state-space. The *Q-value initialization* is a method to speed up the convergence of UCT to a promising policy. While at the beginning of UCT nodes are chosen randomly, as long as there are unvisited actions, the Q-value initialization enables a more efficient discovery of unknown states and rewards. Initializing successor nodes via action-state function assign their estimated reward and lead to targeted browsing of promising areas in the search tree. At last there is a method called *reward locks* to improve the search process in exploitation furthermore. Keller and Eyerich describe the term as the assignment of states, where no action leads to a successor state with a reward greater than the reward of the current node.

Definition 3 *A set of states $S^l \subseteq S$ is a reward lock with reward $r^*(S^l)$, if $R(s,a) = r^*(S^l)$ for all $s \in S^l$ and $a \in A$*

In this way a rollout can be aborted when recognizing a reward lock, where the steps to the horizon are labeled with the reward estimated in the reward lock state. This helps to save operating time. Unfortunately the detection of reward locks is rather hard to perform and uses therefore an approximation method based on Kleene logic (Kleene 1952), that does not lead to an optimal solution but is sufficiently accurate.

2.4.2 PROST - 2014

Unlike its predecessor, PROST2014 is based on the *Trial-based heuristic tree search* (THTS) framework (Keller and Helmert 2013). THTS is a framework to provide several approaches of commonly implemented techniques when solving MDP's. These components consist of heuristic function, backup function, action selection, outcome selection and trial length. As well as in the previous section the search tree consists of decision and chance nodes. To determine the most promising action the state-value function establish the expected reward in a certain chance node. As already mentioned, *trials*, i.e. rollouts, are executed until either an optimum has been detected or the time limit has been exceeded. The individual

stages correspond to the same as MCTS, but using the denoted ingredients as respective decision making function, e.g. a certain heuristic function to initialize successor nodes adding a heuristic value. In addition, there is a fifth component called *recommendation* function, which provides the final action suggestion when the search is finished.
PROST2014 replaces the standard UCT algorithm with UCT* and improves its decision making by concentrating on determining a good decision in the current state. As the environment is defined as MDP, for increasing depth of rollout steps the uncertainty increases due to the probabilistic transitions. Therefore UCT* spends more time in exploring and exploiting the state-space near the current state, instead of examine distant state-action pairs to find a complete policy, by limiting the trial length. When exploring such a bounded tree the rewards of the leaves are determined. If one of the next inner chance node has the information of all successor nodes and their probabilities are given, its total reward can be computed. According to this, a decision node where the entire following chance node rewards are known is labeled as *solved*. A solved decision node provides the best action by choosing the maximum of the chance nodes. With increasing number of simulations, this process repeats. When finally the root is labeled as solved as well, the optimal policy for the corresponding depth is known. The previous components remain part of PROST2014, such as reward locks or determining reasonable and redundant actions.

2.5 PROST-DD

PROST-DD (Geißer and Speck 2018) is an extension of the described PROST2014 planner and is therefore based on the THTS framework as well. By adding further methods the planning system is modified to enable policy search for optimal solutions. To do so PROST-DD applies *Backward Symbolic Search Heuristic* (BSS) as heuristic instead of the originally embedded iterative deepening search (IDS) procedure. As the name implies BSS draws the benefit of symbolic search in MDP's represented as compact decision diagrams. For this purpose Geißer and Speck use the *Algebraic Decision Diagrams* (Bahar et al. 1997). Decision diagrams are directed acyclic graphs, where each node has two childs. A node is representing a variable extracted from the MDP and the edges characterize them as either true or false. The BSS consists of three steps to perform the symbolic search. At first, it has to determinize the MDP, which means to adress actions according certain properties, e.g. the most-likely effects. When initializing the ADD this effect is used as root node, whereby less likely effects are defined as child nodes. The leaf nodes are assigned to sinks with value zero or one. Visualizing all possible combinations of variable settings enable the

representation of transitions of MDP's. In addition, the sinks are replaced by the related rewards obtained of the reward function. After deriving the ADD, the symbolic backward search is applied, which applies as many backward steps as the horizon h is defined. Each step creates a symbolic layer L_i, wherein the maximal reward is stored that is possible to accomplish in the steps that are left over. Subsequently, the determined reward can be associated to the corresponding state and action. Geißer and Speck use therefore following equation to calculate the Q-value of the state-action pair, where i denotes the number of remaining steps according to the horizon h and $S'_{s,a}$ the set of all possible successor states under action a.

$$Q^{init}(s,a) = \frac{\sum s' \in S'_{s,a} L_{i-1}(s')}{|S'_{s,a}|} \tag{2.11}$$

Due to this averaging over the expected rewards of all symbolic layers using action a the backward search can determine the total reward on the given horizon h. Doing so for all possible actions $a \in A$ extracted of the MDP the optimal policy π^* can be established with respect to the horizon. However, Geißer and Speck designate a crucial issue of this approach. Since the state-space might be very large, the search of all maximum expected rewards might be intractable and therefore lead merely to incomplete assignment of the state-action pairs. For this reason PROST-DD makes use of the original setting, when combining the BSS with the iterative deepening search heuristic (IDS) of PROST2014. The search depth d depends on the current MDP domain and its transitions. Thus, if the amount of symbolic layer of BSS becomes too large, both heuristics are combined. Then the IDS performs a depth-first search upon the latest layer computed of the BSS and forwards its estimated expected reward to the Q-value of the layer entries. In the worst case scenario even the first layer is too large to be captured, whereby the planner velves solely on the original IDS heuristic.
PROST-DD participated at the IPPC 2018, but unfortunately only had slightly improved performance compared to its predecessor PROST2014. Geißer and Speck attribute this to the fact that often the tasks were very complex and therefore the domain became too large. In addition, PROST-DD was superior to the other planner system participants when the use of BSS was available.

3 Increasing Horizon Policy Neural Networks

This section points out the purpose of policy NN in finite-horizon MDP's. Furthermore, the procedure how the models are trained is highlighted. This includes the training data preparation as well as the applied algorithm to create and fit the NN's. The general structure and components of the NN are also explained in more detail.

3.1 Intention

Search algorithms such as Iterative Deepening Search (IDS) provide a reliable strategy when searching for a goal in some state-space, by creating further search nodes. IDS provides an optimal solution, but only it requires sufficient resources of time and space. Consequently, search becomes expensive according to increasing search-space. PROST-2014 applies UCT*, which performs expansion and simulations. For large MDP domains the planner is affected by the same bottleneck. Although useful optimizations are implemented which mitigate these shortcomings, it becomes even more difficult to explore sufficient nodes in bounded execution time to provide reasonable results. For this reason we aim to overcome this limitation by using NN's as heuristic. While UCT* has to explore the entire search tree to accumulate the actual Q-value, NN's are trained using training data in terms of state-action pairs and their associated Q-value. Once the NN is sufficiently trained we expect to provide an approximation of the Q-value of a certain state and applied action, even if it was not included to the training set. In the best case, states can be evaluated without having passed through them before and having to calculate their successor states. However, there are some problems that need to be considered. On the one hand, such a training set has to be prepared in advance, which includes generating a sufficiently large amount of training data and defining a reliable method for evaluating the accuracy of the network. On the other hand, training the NN might be time-consuming if a large amount of samples are needed. In the worst case, we cancel out the time advantage achieved from the prediction step over the frequently used application and exploitation.

3.2 Training Procedure

A commonly used approach to train an NN in order to provide the actual Q-value is known as *Deep Q-Learning* where reinforcement learning is used. In Reinforcement learning procedures, the agent or model interacts with the environment by applying actions and is reinforced depending on the outcome, or in other words it is rewarded for a good behavior.

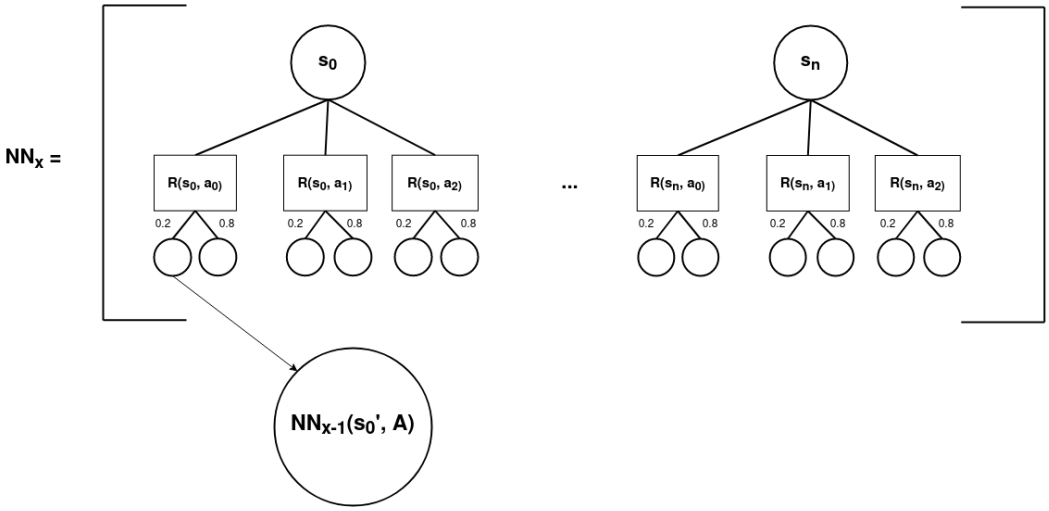

Figure 6: The NN of depth 2 has the information content of $R(s, a)$ and the probability weighted maximum reward of the respective successor. The next NN includes $R(s, a)$ and the probability weighted prediction of the previous network NN_2 of the successor states.

In our case, it is a supervised learning method because the training data is produced and labeled in advance and fed to the NN. This approach belongs to a classical task of ML, which is a so called regression problem. In a regression problem there is an unknown function that has to be determined by passing individual sample points to the NN as training data. With an increasing number of sample points, the NN is able to map the function more accurately and converges in the best case close to the target function. In this way, it should be ensured that even target values that have not been considered can be approximately reproduced. In case of the MDP domain an adequate prediction takes sample points consisting of the current state in combination with an applied action and its label in terms of a reward resulting from this state-action pair.

Since a data point includes the respective state, its applied action, as well as its Q-value, a successor Q-value must be calculated beforehand. However, when generating the data, the effort to calculate a state's Q-value will increase as the horizon increases, causing the same issue as indicated for IDS. For this reason we divide the procedure of determining an NN of horizon d in several steps. First we would prepare an NN of depth 1. However, in our setup such a model would reproduce the reward function, which results in $NN_1 = R(s, a)$. Since the planner already provides this ingredient the predictions of such an NN would be in the best case just as good as this function, but ultimately provides only an approximation. For this reason, we begin to learn an NN of horizon of 2, by exploring nodes up to the depth of 2 and use those results as target Q-value of the training data.

$$Q(s,a)_{d+1} = R(s,a) + \sum_{s' \in S'} p_{s'} * NN_d(s', a)$$

Such a Q-value can be determined by calculating the successor of a certain action in state s and its actual reward using the given reward function, which returns a Q-value for applying a certain action in the current state without regarding the successor state. Multiplying the reward by the probability of occurrence and adding up those results for each outcome of an action a we receive the Q-value of it. The connection of action-state pair and Q-value represents a separate data point and thus enables the training of the NN. Pretending that a network of depth $d = 2$, NN_2, provides perfect Q-value estimations, the procedure is repeated for depth of $d+1$ by calculating the Q-value of depth 1, using the reward function, and adding the result to the prediction of NN_2, that is applied to the successor state of the outcome of the applied action. The joint values thus result in the training data for the next depth of 3 and produce sample points for the following network, denoted as NN_3. Figure 6 illustrates the information content which is included in an NN, respectively the for creation of the training data used ingredients. By successively increasing the depth, a corresponding NN can be trained and only needs to calculate the search depth 1 in addition to the prediction of the predecessor NN itself. Algorithm 1 illustrates the training procedure.

Algorithm 1 provide neural nets of increasing horizon

```
1: def learning(net)
2: trainingData:= {}
3: for s ∈ trainingSet do
4:     actions:= getApplicableActions(s)
5:     for a ∈ actions do
6:         outcomePairs:= expandPDState(s)
7:         reward:= calcReward(s, a)
8:         for pair ∈ outcomePairs do
9:             currentActions:= getApplicableActions(pair.state)
10:            reward += pair.probability * net.predictQValue(pair.state, currentActions)
11:        end for
12:        trainingData.append(s, a, reward)
13:    end for
14: end for
15: deeperNet:= trainNN(trainingData)
16: if timeRemaining() then
17:     return learning(deeperNet)
18: else
19:     return deeperNet
20: end if
```

The process offers a huge saving in nodes to be traversed. The reason for this is that the

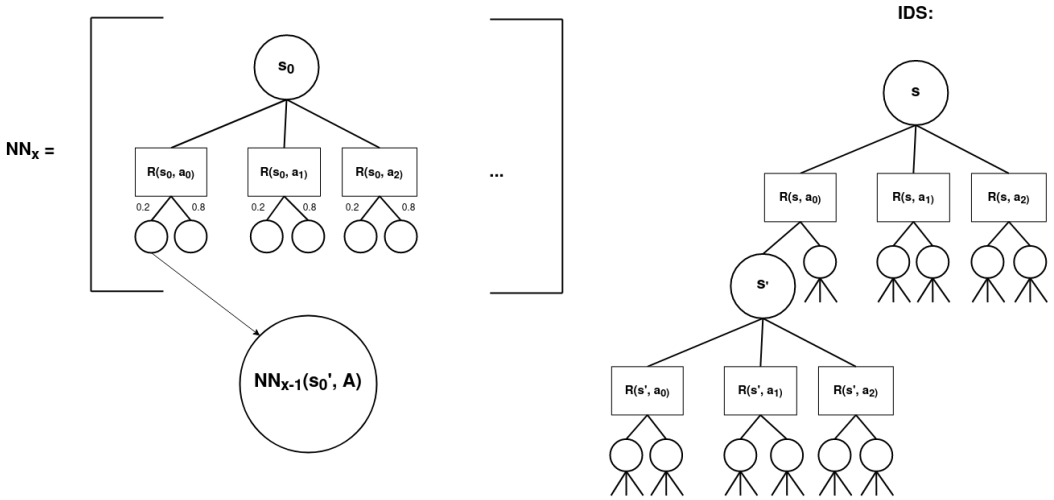

Figure 7: The number of node expansions remains constant for NN training of increasing depth. On the other hand IDS has to perform an increasing amount of transitions when taking the next depth into account.

number of successors remains constant. In contrast, an IDS has to perform an exponentially growing amount of transitions with increasing depth, due to the branching factor after each node. In Figure 7 is a graphical representation of this fact. However, it must be mentioned that in addition to the effort spent on training, the effort of making predictions must also be taken into account. Here, the amount of computational effort depends on the breadth of the NN, as well as number of hidden layer, since with increasing width and depth, both the computation of predictions and training requires more time.

3.3 Learning Components

After explaining the general training algorithm, this section highlights the details of the learning procedure. This includes the NN structure such as the hyperparameters and in particular the subtleties of the training data generation.

3.3.1 Network Components

The structure of NN designates, among other things, the input-, hidden- and output layer. The input layer determines the number of features that a single sample point includes and which are fed to the network. It is important to provide all the necessary information needed to determine reasonable Q-values, which in this simulation includes the current state and the applied action. As mentioned above the PROST-planner defines states in terms of digit sequences. Thus, the input layer size results from concatenation of state and action

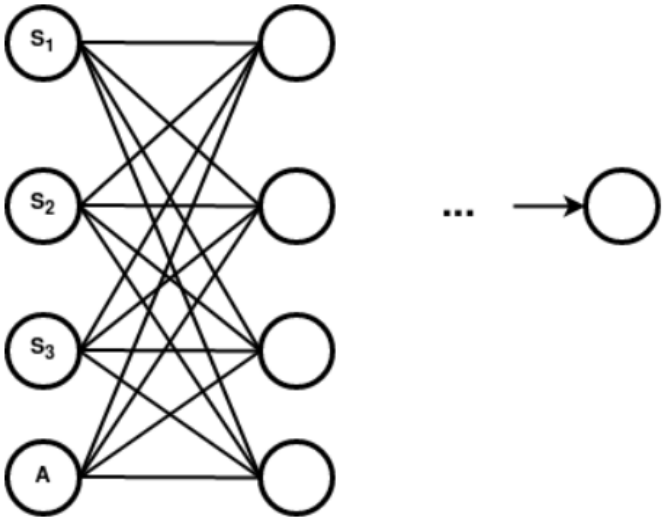

Figure 8: Structure of a Q-value network where the input neurons process the state variables and the current action index. As a result we receive a corresponding Q-value.

sequence. A simple model of such a Q-value NN is depicted in Figure 8. Finding a reasonable number of neurons in the hidden layer is difficult to determine at first. The most promising breadth and amount of hidden layer will be tested in several experiment trials. In the initial experiment of increasing horizon NN using the previous trained network to provide training data for the following depth, it is sufficient to define the output layer to the size of one, since each sample point should lead to a single Q-value. The core of train NN consists of the so called optimizer. As the name suggests, the optimizer enables optimizing the function described by the current NN predictions, which aims to describe the reward function. For this purpose the neuron weights are adjusted depending on the optimizer. In this experiment we are using the stochastic gradient descent (SGD) optimizer.

Before going into detail about SGD, we introduce another component that has an significant impact on the optimization. In the previous chapter we introduced the loss function which designates the handling of the difference of predicted and target values. In the following implementation, we mainly consider two particular variants to solve the regression problem. The *Mean Absolute Error* (MAE) or *L1 loss Function* calculates the absolute difference of predicted and target value.

$$l_1(x, y) = |x - y| \tag{3.1}$$

A further variant is the *Mean Squared Error* (MSE) or *L2 loss*, which determines the

average difference of predicted and target values.

$$loss_2(x, y) = (x - y)^2 \tag{3.2}$$

Both calculations are frequently used in regression problems. Each has advantages as well as disadvantages over the other. The L1 loss enables the reaction to individual boundary values such as much greater or smaller than the average value distribution. Since the L2 loss squares the difference large erros are punished harder and errors smaller than 1 are mitigated. Each loss function will be applied in order to determine whether the L1 or L2 loss produces better results.

The SGD optimizer is used to minimize the cost, calculated by such a loss function. The iterative optimization is similar to the gradient descent described above. However, gradient descent takes the whole set of training data into account when performing the algorithm. This is the reason why this procedure is also called batch gradient descent. In case of SGD only one data point per adjustment is taken into account and subsequently all other sample points are applied to update the current weight. There is a hybrid of both optimization functions, where the training set is divided in several subsets, known as mini batch SGD. There is an extension of the simple gradient descent algorithm. The so called *momentum* (Murphy 2012) is an additional factor that decreases the impact of previous weight adaptions. Since the SGD does only take a subset of training points for each updating into account, we expect some noise occurring in the intermediate training progress. The error is noticeable by the erratic rise and fall of the loss, as seen in Figure 9. To overcome this behavior, a term is added to the gradient descent Formula 2.9. The term includes the last adjustment of θ_{k-1} and thus shifts the current value slightly ot the original position. The parameter μ is set in the range of $[0, 1]$ and determines the impact of the momentum term.

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t) + \mu_t(\theta_t - \theta_{t-1}) \tag{3.3}$$

The extension of the algorithm has the consequence that the intermediate results of SGD deviate less from the gradient descent variant.

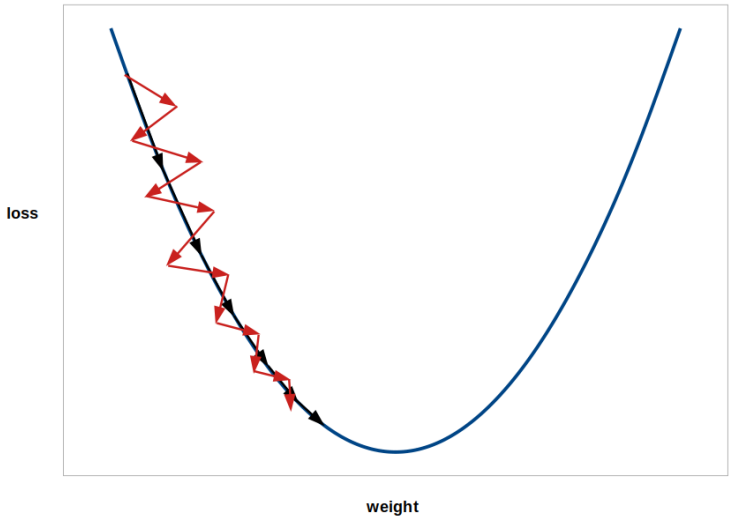

Figure 9: The stochastic gradient descent, depicted as red lines causes wider outliers than the normal gradient descent, represented in black lines.

3.3.2 Training Data Preparation

In order to prepare the training data it takes many different states of a certain instance environment to represent a large part of the target function as accurate as possible. PROST provides a training set, which includes by random walks generated states, which can be used as initial entry point when creating training data. However it must be noted that such training states are not necessarily representative in terms of information content. For example, in the *sysadmin* instance, the goal is to keep as many of a selection of computers turned on as possible. If all machines are already turned off bad decisions have already been made in advance. How well the model performs in such a miserable situation is therefore meaningless.

As shown in Algorithm 1, the applicable actions are determined and applied to the respective state. Since MDP's have the property to be non-deterministic, an applied action can end up in several successor states under some probabilities. When expanding a certain state all outcomes are evaluated and weighted with the probability of occurrence. The sum over all probability weighted outcomes gives the expected reward of the state-action pair. Unfortunately there are some domains with very large amount of possible successors, which makes it intractable to take all of them into account. For instance, harder instances of the *sysadmin* domain leads to 2^{50} outcomes for all state-action pairs. To overcome this limitation, the number of expanded nodes is limited to some *threshold*. If the boundary is

exceeded there are just a part of the entire set of possible outcomes sampled. The successors generated by the *sample()* function, in Algorithm 2, are created in dependence of their probability, i.e. more likely samples are drawn with increased frequency. Although not all successors can be calculated, we assume that by a correspondingly large number of sampled states, the most important, respectively most probable outcomes are explored.

Algorithm 2 expand state if threshold is observed

```
1: def expandPDState(s)
2: outcomePairs:= {}
3: if state.getNumberOfPDSuccessors() < threshold then
4:     outcomePairs:= expand(s)
5: else
6:     outcomePairs:= sampleOutcomes(s, threshold)
7: end if
8:
9: def sampleOutcomes(s, numberOfSamples)
10: accumulatedProb:= 0.0
11: outcomes:= {}
12: while outcomes.size() < numberOfSamples do
13:     sampledState:= s.generateRandomOutcome()
14:     outcomes.append(sampledState)
15: end while
16: return outcomes
```

3.4 Optimizations

The ingredients introduced so far already provide a general approach to generate a heuristic. However, are still necessary adjustments that need to be considered in more detail. In this chapter, we specify the adaptions and demonstrate how they are implemented.

3.4.1 Mini-Batch

Despite the limitation of generating of sample points to the depth of 2 for each training data preparation, very large amounts of data are generated, since each combination of state and action corresponds to a single input. Passing each element individually also means that a separate optimization is performed using backpropagation. The training usually runs through several epochs, i.e. the same data set is fed into the network several times. This ensures that the weights are sufficiently attached and that the function explains the samples well. If the input quantity is already large and many epochs are required, this can lead to a correspondingly great investment of time. For this reason, data points are passed in *batches*, which means that first all features are fed to the network and an average total loss

over all outcomes is calculated. Subsequently, a one-time optimization can be performed with the help of the average error rate. However, using batches still takes a lot of time, because all results have to be saved until the backpropagation has been executed. To avoid this problem, the total amount of elements can be divided into mini-batches. This way, an average of a smaller number of training data is used per optimization step and requires only a fraction of the memory used when applying a full batch. In different domains, the number of state-action pairs can vary greatly, so a general might become very difficult. Therefore the overall optimum must be evaluated by extensive testing several batch sizes. It turned out that even with the changeover to batch-wise handling of the training data, the generation of all NN takes an enormous amount of processing time. Since PROST is actually used in the IPPC competition, the learning procedure to explore the environmental behavior and determine the final path is limited in time. The generation of NN's takes too much time and is therefore aborted before a result could be generated. In order to ensure that a result is delivered, we separate the training procedure of the networks and their parameters, such as weights or bias, from the actual search step. In the following we will designate the procedures *generation* and *evaluation* run. In this way, an increased number of epochs can be used in the experiment at a lower learning rate to ensure improved accuracy.

3.4.2 Speedup Convergence of NN

In our approach creating an NN of each depth, a new initial model is created in each iteration and has to be trained. When creating a new network, the weight of each neuron has to be initialized by some value. In PyTorch the default initialization depends on the used type of layer, in the case of linear layers these are initialized as:

$$\frac{1}{\sqrt{input\_features}}$$

For this reason the initial prediction of a new generated NN is rather small. As long as the target values themselves are still comparatively low as well, the gradient can be calculated with the help of the loss function and the weights can be adjusted on the basis of this. In MDP's it is often the case that the rewards increase with deeper horizons, as the respective rewards are accumulate up to a certain depth. The resulting difference between initial NN predictions and the actual value there continues growing as well. Considering the loss function, we see that the loss can become very high even with a small difference, since the result is obtained by squaring. In this way, the distance to converge becomes even longer by increasing search depth model preparation. In order to avoid the steady increase, the

respective generated NN must be initialized with suitable default weights. In general, it is difficult to make a blanket determination of which values are suitable for this purpose, as the range of reward in a certain depth can vary greatly depending on the given task. However, we assume that the reward of the last generated NN, i.e. the net of a depth one less then the current, usually generates close predictions to the new target. Since the following sample points depend on the prediction of the previous net and the result of the reward function, the subsequent outcome does only differ by the calculated reward of the reward function. Hence, as an alternative to the NN with default weights the previous net is used as a new base when learning the corresponding depth. Another significant advantage of this method, besides accelerating convergence, is that it reduces the probability of getting stuck in a local optimum. This confirms our assumption that the previous NN serves as reasonable default value of the current.

3.4.3 Caching

Another adjustment is made in the caching behaviour of the planner. PROST stores both the action itself and the outgoing state when determining the applicable actions. Although already the successor states of a certain action bounded by the threshold allows only a certain number of successors, with each pass under a certain probability different successors are generated. This leads to the fact that in spite of limitation constantly new unique pairs of outcome stat and applied action arise. For domains with a large number of actions and a wide split of possible successor states, the memory overhead exceeds the capacities. To avoid overloading the system resources, the memory used for caching is monitored during the generation of new sample points and caching is disabled if necessary. Thus, the generation phase will take additional time and validates the encapsulation of the evaluation.

3.5 Policy Network

To train the Q-value network, we generate state-action pairs and feed the data into the NN. The randomly created training set consists on average of 100 up to 250 states. The generation of the NN of a single domain can require a lot of time depending on the amount of applicable actions as well as the horizon length. However, there are tasks whose number of applicable actions makes it intractable to calculate all their corresponding outcomes. One possible approach would be to train only a certain subset of the actions, but this raises the question of which ones should be emphasized and which ones should be neglected. PROST already provides a mechanism that checks whether different actions achieve the same result. So it

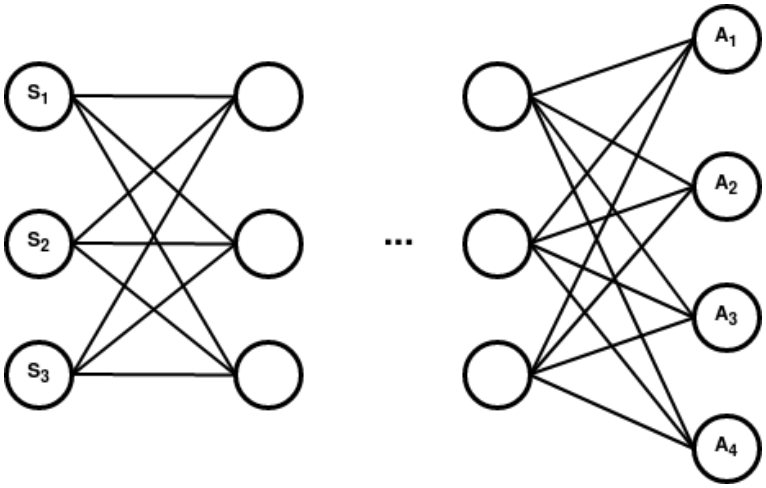

Figure 10: Structure policy network; input neurons take state variables and output Q-value for each action index

can be assumed that all actions declared as applicable are unique in their state influence. A meaningful classification into considered and neglected data points is therefore hardly possible without including more detailed information about the current task.
To overcome this constraint we introduce another variation of the Q-value NN. The *policy network* predicts a Q-value for each action applicable in the current state. Instead of just providing the Q-value of a single state-action pair, the model conveys the best action in a given state. Thus, the training preparation changes as well, since the input simply consists of a state, while the output becomes a q-value for each applicable action. Although the amount of training data is still the same, the number of sample points is drastically reduced in this way, since only one sample per state in the training set is needed. This in turn results in an enormous saving of computing time. The policy, which should return the best action, can be determined afterwards by choosing the maximum value of the outcome sequence. The neuronal arrangement of the NN is illustrated in Figure 10.

In addition to the advantages, it must be taken into account that the modified network also brings drawbacks. Thus, the result of a prediction will output a value for every action without exception. The behavior is not an issue as long as all actions are applicable. However, the inapplicable ones cause a problem that needs to be specifically addressed. In the evaluation phase all unnecessary entries can be ignored, but in the generation phase the question arises how the learning approach should turn out if individual values are to be ignored. The loss function always calculates the difference between prediction and target value. On the one hand, there is no q-value for non-applicable actions in the environment.

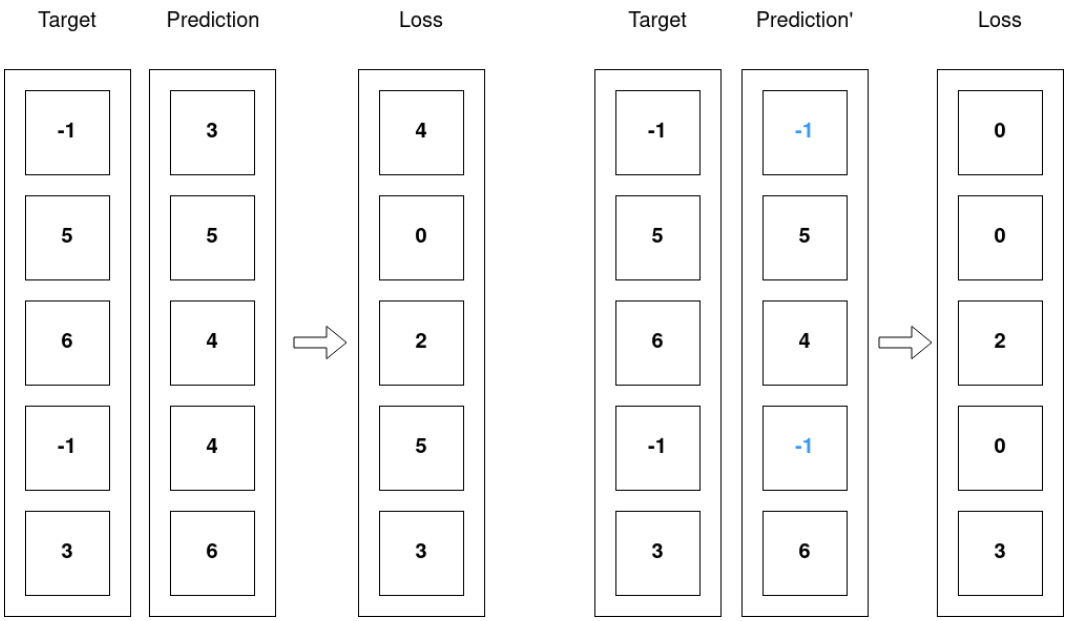

Figure 11: Prepare prediction to adjust the L1-loss $loss(x, y) = |x - y|$ by pasting -1 in prediction tensor

On the other hand, even if there were such a target value, there should be no adjustment based on the loss, since the entry will not be considered by the planner anyway. A suitable solution is the adaption of the target, as well as the prediction sequence. When generating the training data, the q-values for non-applicable actions are set to -1. After the prediction tensor is generated, the entries of the target tensor are checked for the manually set values and matching elements of *i-th* position are copied to *i-th* position of the prediction tensor. Since, as described above, the reward function has been shifted to a positive value range it is executed that an applicable action has caused the value -1. By equating the target and prediction at *i-th* index, the corresponding calculated loss becomes zero and does not cause any adjustment to the weights of the model by itself.

4 Experiment Setup

This section details the experiment settings, including the choice of hyperparameters and learning procedure in order to provide a heuristic. Since both the Q-value network and the policy network operate very differently and follow a different approach, both variants are treated as independent experiments. As already mentioned, the process of creating networks and actually searching a plan using the provided predictions, is divided into separate procedures. Before elaborating on these, we first examine one of the most essential ingredients in our experiment, the training set.

4.1 Enlarge Training Set

In the experimental setup we apply the NN procedure to first 10 instances of the IPPC benchmarks of the years 2011, 2014 and 2018. In total, with a number of 20 domains, this results in 200 experiment instances. For each of these instances, PROST creates a training set by simulating random walks beginning in the initial state. Depending on how many unique states are discovered, the size of the pool varies from about 100 to a maximum of 250 states. This amount may be sufficient for little domains that have only a small state-space. For most tasks in our experiment, this factor is far from adequate. This is shown in previous performed tests and is not surprising, after all, the network has to learn the most important relationships of the state variables. A huge state-space of thousands of different dependencies between the state variables is therefore unlikely to be represented by a training set of merely 250 examples. In general, one can say that an NN can only learn the correlations of a certain MDP that are also represented in the training data. Thus, depending on the states explored by random walks, the results would be determined by chance.

To meet this intention, methods are investigated to increase the pool of possible states. In the most desirable case, the NN maps more accurate Q-values to the fed states. On the one hand, further states can be determined with the help of the PROST planner itself. So far, a limited number of random walks is executed at the beginning and the resulting states are recorded in the training set. By increasing the number of applied simulations, more possible states are reached and thus increase the dataset. Depending on the probability distribution of the individual states of the respective MDP's, rather unlikely actions are not explored however, this approach gives a cross-section of the most likely states. However, states created in this way are not necessarily meaningful states. The quality of the training

data is very domain specific, because the state variables have very different meanings. In the *sysadmin* domain, the aim is to keep as many computers as possible switched on, which are going to be deactivated by chance. Due to the random walk it can happen that many actions are performed that do not keep the computers running. If the agent finds itself in a situation where only a few computers are still on and are switched off again after being reactivated, it is possible to learn the best action in such a scenario, but up to this point a lot of bad decisions have already been made. In this are of the MDP state-space there are a lot of states, but they contain hardly any qualitatively interesting information for the training of the model.

On the other hand, the whole set of potential states can be identified by expanding from the initial state using all applicable states. All new states, i.e. those that have not been discovered before, are collected and all possible actions are performed from them as well. The general procedure can be taken from Algorithm 3.

If we were able to compute the total state-space for all domains, we could also compute rewards of promising paths. Such a brute-force approach, which can be denoted as a *breadth first search*, is thus generally inappropriate for planning tasks and contradicts the very idea of planning itself, which consists of avoiding unnecessary effort with a deliberate and well directed approach. Nevertheless, this experimental setup gives us an insight into the theoretically achievable results. In addition, such a trial will proof, whether with a sufficent amount of training states the PROST-2014 planner can be outperformed. Even tough we cannot fully determine all domain specific data sets there are some instances that are suitable for this purpose.

4.2 Q-value Network

As a reminder, this kind of NN obtains its information from the respective training states including the action applied in terms of an action index. The output describes the Q-value from the state-action pair. The appropriate choice of hyperparameters is difficult to determine in advance. For this reason, several experiments with a wide range of parameters are used to draw conclusions about better values for them. Hence, the batch size is set on 1000, 2000 and 5000. The values are chosen rather high, since the number of samples is defined by $numberOfApplicableActions * numberOfSamples$, becomes very large for some domains with many different actions and successors. Such a big training set increases the duration of a learning cycle. Due to the size of the batches, more examples are applied at once and thus reduces the time spent to perform the training. The learning rate is set to the

Algorithm 3 discover all possible states

```
1: def discoverStates()
2: stateSet:=empty()
3: statesToExpand.append(initialState)
4: while not statesToExpand.empty() do
5:     next:=statesToExpand.pop()
6:     if next ∉ stateSet then
7:         stateSet.append(next)
8:         expandState(next, statesToExpand)
9:     end if
10: end while

11: def expandState(s, statesToExpand)
12: actionsToExpand:= getApplicableActions(s)
13: for a ∈ actionsToExpand do
14:     successor:= calcSuccessorState(s, a)
15:     statesToExpand.append(successor)
16: end for
```

values 0.01 and 0.001, which are a frequently used default parameter when training different types of NN. Whether a larger selection of parameters is necessary will be evaluated later. An equally significant influence is the number of epochs. If too few iterations are used, it may happen that the NN has not yet been sufficiently trained or in other words did not converge. So a naive approach would be to perform a lot of epochs, though then the training set is described very precisely, but in contrast states outside of the samples very poorly. This process is called *overfitting*. To get an idea about the influence of this setup value, the epoch number is set to 10, 50 and 100.

Furthermore the network breadth and depth must be defined. Determining the optimum ratio of width to depth of an NN is still a present problem (Stathakis 2009). Although Stathakis deals with a classification problem, this is very similar to our supervised learning intention. In his article *"How many hidden layers and nodes?"*, he discusses several approaches that allow the optimal net dimensions to be approximated. This question itself represents the scope of an entire paper, which is why in this thesis only an approximation and not an optimum will be found in order to draw conclusions about the gained value of the learned heuristic process. After Stathakis, we will use trial and error to find the most suitable size ratios that we can approximately determine. First of all, we would like to get a first impression wherefore it is enough to train small nets. In this experiment it does not make sense to set a fixed size for the breadth of the NN for the different domains. Since the input depends on the number of state variables used to describe the current states, the input layer size varies and is therefore spread to a very breadth hidden layer or compressed

to a narrow one depending on the task. To avoid this, the breadth is set to the size of the input layer and multiplied by a factor. So when we talk about an NN breadth of 1 in this work, the factor is meant and provides an actual number of neurons equal to the input layer. Therefore, the depth of the NN, i.e. the number of hidden layers is set to 1 and the breadth to 1.

Finally, we define the parameter of the *threshold*, denoted in the previous chapter *Training Data Preparation*, which specifies the number of states that should be expanded given the probability distribution of the current state. The *threshold* thus determines the accuracy of the target value when a particularly large number of possible successor states follow an action. First tests showed that already at a *threshold* of 50, some domains like the *chromatic-dice* variant have too large memory demands. For this reason, the experiments are initiated with a comparatively small limitation of 10.

4.3 Policy Network

Since the structure of the policy network differs from the Q-value network, which is exemplified in figure 10, the hyperparameters must be chosen differently. The amount training data of the policy network is much smaller than the one of the Q-value network. Since the input consists only of the state variables, the number of training states corresponds to the number of inputs fed to the NN. A smaller amount of training data also allows us to perform more epochs. The values for this parameter range are distributed over 10, 50, 100 and 1000, to clarify whether more epochs are worthwhile. Batch size and learning rate remain the same such as in the Q-value network setting. Likewise, the dimension of the network is set to a breadth and depth of 1. Since the breadth already dynamically adapts to each task, it is excluded from the parameter search.

4.4 Generate procedure

The process of generating the NN of depth equal to the horizon is very time-consuming and therefore performed in advance in the case of both NN types. Although care was taken to allow significantly more computing time for the experiment, not all NN's are calculated. The reason turns out to be that the tasks with particularly many actions could not be produced, as they need noticeable more time when generating NN's. Especially domains like academic-advising-2018 instance 10 stand out with almost 2000 actions, but also domains like academic-advising-2014 instance 6 with 211 actions reached only a $NN_3 1$ given a horizon of 40. Without a network for each depth the evaluation procedure does not provide a result

for such a task in the evaluate procedure. Fortunately due to the restriction of the *threshold* parameter only a few tasks are not able to generate the full network set. Therefore 189 of 200 tasks can produce an average reward. Of course, it is essential to provide some form of evaluation whether a trained network is sufficiently accurate in representing the correct Q-value according to the fed data. For this reason we introduce the $evaluateNetwork()$ function. Therein the L1 loss or MAE is determined using the total amount of samples as batch size. Thus, the mean value of the absolute error is determined and shows how well the trained set is mapped to the heuristic. To determine the accuracy of a generated NN of unssen data, we introduce yet another evaluation option in this step. For this purpose, we divide the training data into a test and a train set, where the test set contains 10% of the total sample points and all the others are assigned to the train set. Only the data points of the train set are fed to the NN, whereby the test set contains data that have not yet been learned by by the model. The accuracy of the two data sets is determined by the L1 loss over all data points of the respective set. Thus, we can determine how well the model can describe the trained data and whether even unknown data achieve sufficient accuracy.

4.5 Evaluation procedure

Most time of the preparation takes place in the generation procedure. After training, the parameters of the NN's are stored and managed in folders sorted by its task. Subsequently, these files can be loaded in the evaluation procedure. This approach results in an enormous time saving during the search run, since the learning phase has already been completed. It should be noted that the network must be initialized with the appropriate dimensions, which means the size of input hidden and output layer, as well as the the number of hidden layer. Since only the weight and bias are stored, it would lead to wrong predictions if not all neurons are mapped in the same way as before. There are several instances which do not achieve the maximum number of NN. To avoid that these circumstances lead to no result at all, we use for deeper levels than NN's determined just the deepest NN. Pretending that this network performs the best actions, with a horizon equal to its depth, this approach could still lead to good rewards. Moreover, we must note that in the case of MDP's of it, the results are random. To obtain the probability of individual outliers in the results resulting from chance, the search is performed 100 times and the average reward of all individual runs is calculated.

5 Parameter Determination

In this section we aim to determine reasonable parameters for the NN to achieve in the experiments as good results as possible and optimize the planner. Distinguishing all possible combinations is unsustainable in terms of computational effort. For this reason, we compare a small set of values that seem reasonable to get at least a rough estimate of which settings are appropriate. As discussed in the previous section, we have determined a setting that provides a first clue. Once this has been evaluated, we select each parameter individually by choosing values in a certain range where we expect some local optimum and set the appropriate value for a final setup. With these settings we create another model that allows us to make comparison with PROST2014. Furthermore, it should be mentioned that the following experiments use the MAE loss as well as the SGD optimizer.

5.1 Rough Alignment of Default Parameters

The results of the defined parameters are illustrated in the following table, where in each case only the best average reward that could be determined is chosen. The first three columns refer to the best results for different number of epochs. The right columns depicts the best values for each batch size that occurred. The so called *ipc-score* is used as a guide for assessing the experiments. The ipc-score is calculated by the ratio of the current and the best trial over all of this run. This results in an ipc-score of 1 for best run. In addition, there is a defined minimum reward for each instance. If the average reward falls below this minimum, the corresponding ipc-score becomes 0.

$$ipc - score = \begin{cases} \frac{current\_reward}{best\_reward}, & \text{if } current\_reward > min\_rewards. \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

Although there is also an *average-reward*, which calculates the average reward achieved over alle runs in a certain instance. It seems to be a more appropriate variable to assess model parameters, however the rewards of the individual instances are widely spread and instance respective rewards are not normalized to each other. For example the instance of *push-your-luck* rewards very risky actions with a comparatively high payoff for success, while the outcome is very small for loss. In comparison, the *elevator* domain causes in general a rather low reward and therefore would have a much weaker impact on the average-reward. Therefore the average-reward can be consulted for a rough estimate, but the actual decision is made on the basis of the normalized ipc-score.

The Table 1 of Q-value NN's clearly shows improving results for increasing number of epochs. Here the row of average-rewards designates the sum over all averaged rewards in each Domain. The outcomes reveal that too few epochs negatively affect learning success. In addition, it is probably interesting to perform further training runs with even more iterations, since we have not yet established an upper limit.

Q-value	NN-setup_1	NN-setup_2	NN-setup_3
average_reward	-150,888.92	-111,314.84	-87,498.49
ipc_score	119.25	125.82	136.43
epochs	10	50	100
batch size	1000	1000	1000
learning_rate	0.001	0.001	0.001
hidden	1	1	1
breadth	1	1	1

Table 1: Best results of different epoch values using the Q-value NN

In contrast, as Table 2 suggests, larger batch sizes do not seem to be more successful and even might have a negative influence. However, we will examine a wider range to ensure that no better setup with increased batch size exists near to our local optimum. Furthermore, we notice that among the top results only a learning rate of 0.001 appears. The step size of 0.01 is therefore too large and does not converge to the local optimum. In the same way as for the number of epochs, a better fitting learning rate value could be promising.

Q-value	NN-setup_1	NN-setup_2	NN-setup_3
average_reward	-87,498.49	-104,917.61	-126,419.93
ipc_score	136.43	128.75	126.89
epochs	100	100	100
batch size	1000	2000	5000
learning_rate	0.001	0.001	0.001
hidden	1	1	1
breadth	1	1	1

Table 2: Best results of different batch sizes using the Q-value NN

The rewards yielded of the policy NN's are listed in Tabel 3. According to the Q-value table, the results behave similarly. It can be recognized that an increased epoch value has a positive influence.

Policy	NN-setup_1	NN-setup_2	NN-setup_3	NN-setup_4
average_reward	-133,315.63	-120573.33	-114,974.03	-112,661.06
ipc_score	98.30	124.53	125.76	129.46
epochs	10	50	100	1000
batch size	1000	1000	1000	1000
learning_rate	0.001	0.001	0.001	0.001
hidden	1	1	1	1
breadth	1	1	1	1

Table 3: Best results of different epoch values using the Policy NN

Also the batch size shows the same behaviour as above, where large values are only of limited success. In addition, the learning rate of 0.001 dominates the best trials. On the other hand, we can see an outlier of the run *NN-setup_2* of Table 3, where the number of epochs and the learning rate differs to the generally dominant parameters. As mentioned above domains such as *push your luck*, where randomness has a significant influence on the outcome, may lead to unexpected outcomes of this kind. In this run we can observe exactly this behavior. It was expected that the experiment with the dominant values in epochs of 1000 and the learning rate of 0.001 would achieve the best result. However, it only achieved an average reward of -121905.23. A closer look at the push-your-luck domain reveals that the outlier has a very high reward of 8911.65 while the favorite only reaches 1462.44.

Policy	NN-setup_1	NN-setup_2	NN-setup_3
average_reward	-112,661.06	-116,454.44	-113,973.71
ipc_score	129.46	128.16	126.71
epochs	1000	100	1000
batch size	1000	2000	5000
learning_rate	0.001	0.01	0.001
hidden	1	1	1
breadth	1	1	1

Table 4: Best results of different batch sizes using the Policy NN

Based on this observation, we will use the dominant values for learning rate and batch size for the following parameter approximations. By doing so, we expect to get closer to a local optimum by approaching appropriate parameters. Adding all the maximum scores of the policy NN's results in a total value of 148.68. Even the rough trial shown here shows a

potential for improvement. This holds as well for the trials of Q-value NN's with an overall maximum ipc-score of 151.06, which is an enormous enhancement compared to the current results.

5.2 Number of Epochs

At first, we explore a reasonable number of epochs which has a great influence on the other parameters and serves as an entry point for further settings. The previous rough assessment have shown that 100 epochs is a reasonable starting point. Since the lower marginal values of 10, 50 rapidly deteriorate in the results, the selection of parameters is chosen above the entry point. Figure 12 illustrates the achieved ipc-score of each setting, where each segment corresponds to an own task domain.

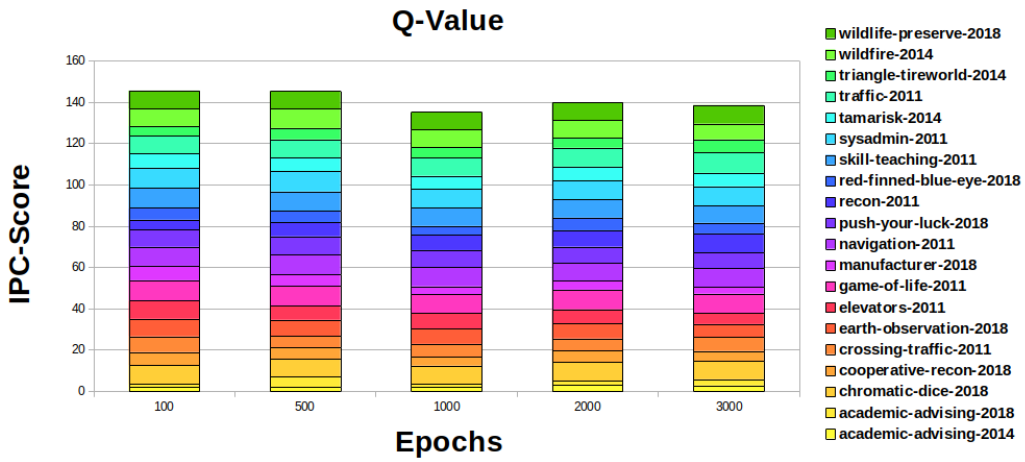

Figure 12: Results of the Q-Value model parameter exploration - number of epochs

We can see that the setting for 100 as well as 500 provide the most auspicious results. In general, we would expect that increasing iterations produce more promising NN, since there are more optimizing steps performed. Nevertheless, the computation time for generating the NN is limited, which means that the respective task does not provide all NN's up to the total horizon. As one can see in the Figure 13, the total number of generated networks decreases with growing number of epochs.

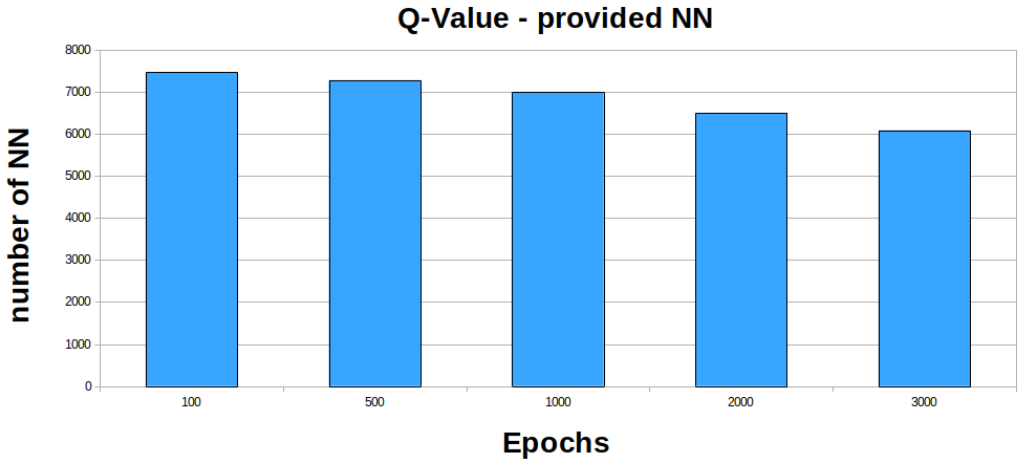

Figure 13: Amount of NN provided in Q-Value model generation procedure for increasing number of epochs

While not all NN can be generated within time anyway, a high epoch number amplifies this effect. In domains such as *academic-advising-2014* and *academic-advising-2018*, not even the half of the number of NN's that are available at the lower epochs can be developed. The lower horizon thus leads to a worse reward. While both 100 and 500 epochs perform similarly, we have only a small loss in the number of NN. In addition, for the learning rate, we may need a higher number of epochs to allow convergence of smaller step sizes. For this reason, 500 is used as the new base value.

The same observations can be made in the case of the policy NN, as Figure 14 shows. Again, a peak, albeit a slight one, can be detected at 500 epochs. Similarly, the number of total generated NN also decreases here with a discrepancy of even more than 1000 NN between the smallest and largest number of iterations. In general, it seems that the number of epochs only has little impact here. However, all settings must always be considered as a whole. So for instance it might happen that the epochs in this setup lead to similar results, but when changing the learning rate a too small number of iterations is not sufficient to guarantee a convergence of the function. Thus, the number of epochs of the policy NN type is also set to 500, so as not to restrict the opportunities of more precise learning rates in advance.

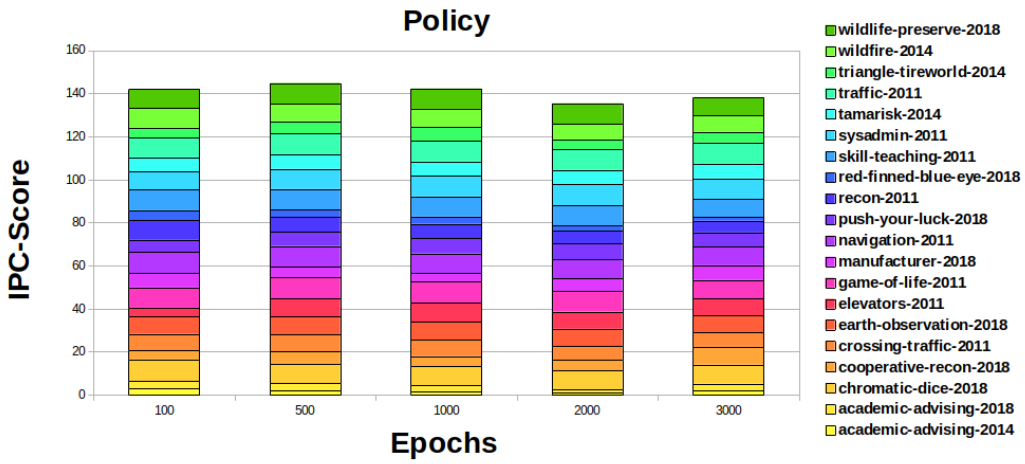

Figure 14: Results of the policy model parameter exploration - number of epochs

5.3 Learning Rate

To determine a suitable learning rate we mainly consider a distribution of finer values, which should allow a more accurate approximation to the target function. Since we produced significantly worse results at a value of 0.01 in the rough alignment at the beginning of this chapter, we do not consider even coarser selections. This assumption is confirmed by the overview of the Q-value NN results in the chart below.

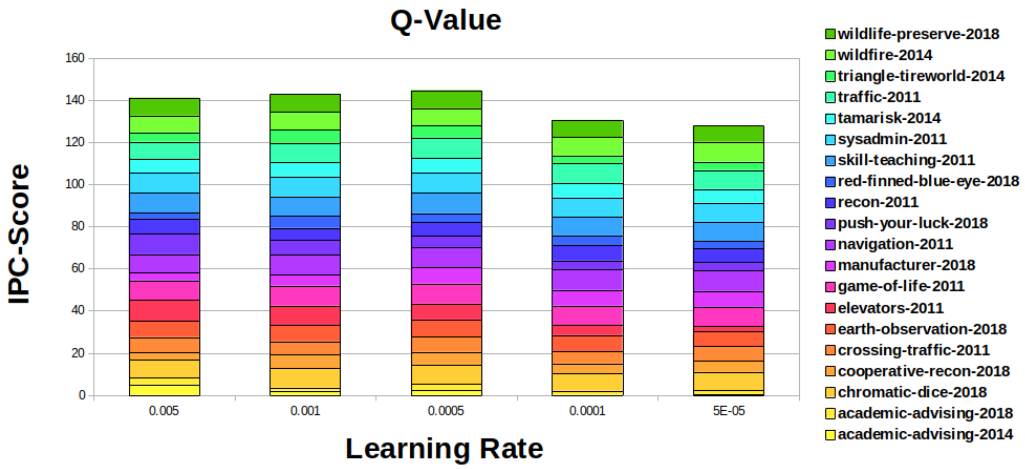

Figure 15: Results of the Q-Value model parameter exploration - learning rate

A slight improvement can be recognized when using a learning rate of 0.001 or 0.0005. The fast decrease of the ipc-score with more precise value ranges shows how sensitive the adjustment of this parameter is. Actually, a smaller learning rate allows a more precise approximation to a target value, but due to smaller step sizes the model also needs more epochs to reach the target. In our case, if we take a closer look at the loss of the test set, we can expect the inaccuracy to increase as the learning rate decreases too much. The diagram below illustrates the average loss of all NN per domain and disproves this assumption. Since many domains show a comparatively very low loss, they are not visible in Figure 16.

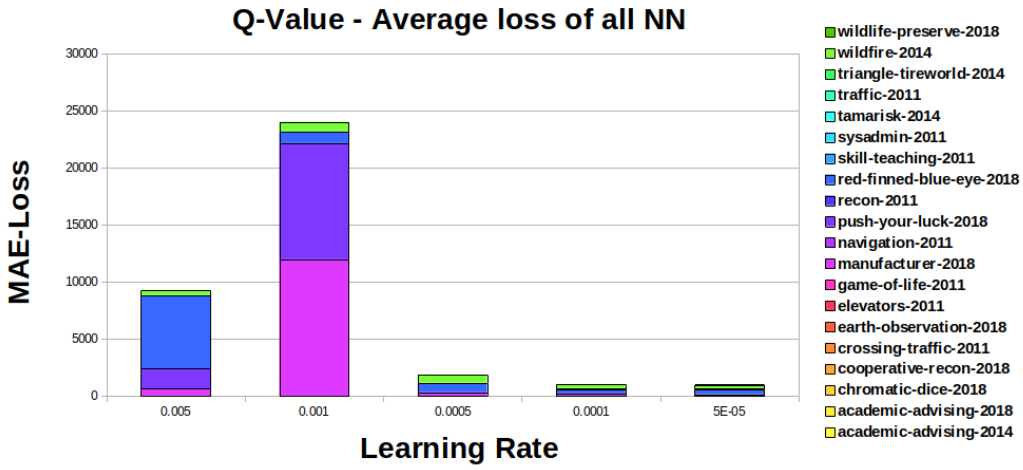

Figure 16: Average loss of Q-value NN's

To get to the bottom of this phenomenon, it is first important to consider how our training data are generated. At first, we are applying the reward function that gives us an outcome for each state-action pair. Next the previous model is used to calculate the Q-values of the successors. If the NN_{d-1} already produces rather inaccurate predictions, the error rate accumulates with each additional NN. Thus, if a model is able to reproduce the prediction of the predecessor, it does not mean that the error rate to the actual target function is low. These circumstances is illustrated in Figure 17. Although the model with the low learning rate is not able to reproduce the initial function, its successor NN easily learns the new training data that depend on its predecessor. Nevertheless, the trained NN is subsequently not able to make good predictions of the real Q-value function.

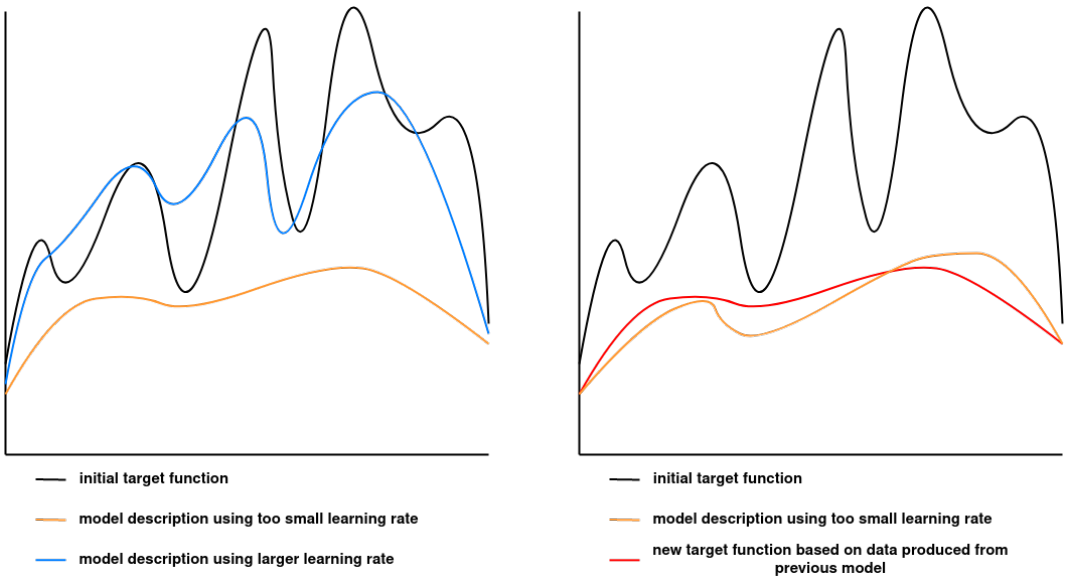

Figure 17: Left shows a certain target function, described by a model using too small step sizes and a model using larger step sizes. Right illustrates the next time step, when the new target function of the orange model is described

In order to classify the loss it is therefore important to look at the loss of the very first NN, since its training data does not depend on any other NN, but only on the accumulated reward function of the current state, the applied actions and the resulting successors.

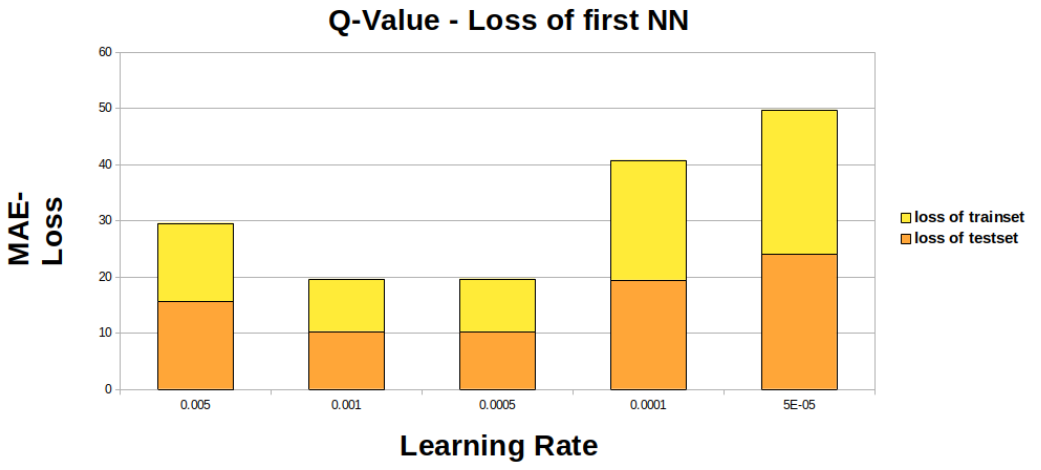

Figure 18: Loss of first Q-value NN

Considering the average loss of the first NN in Figure 18, it can be recognized that this

actually increases significantly with a lower learning rate. Also, if the precision is too low, the training, as well as test data cannot be described as properly.

The results of the experiment with the policy NN behave very similar to the results of the Q-value variant. Again, the best outcomes are shown for the values 0.001 and 0.0005. Since in both experiment setups the more precise learning rate performs better, even if only slightly, it will be applied in the upcoming experiments.

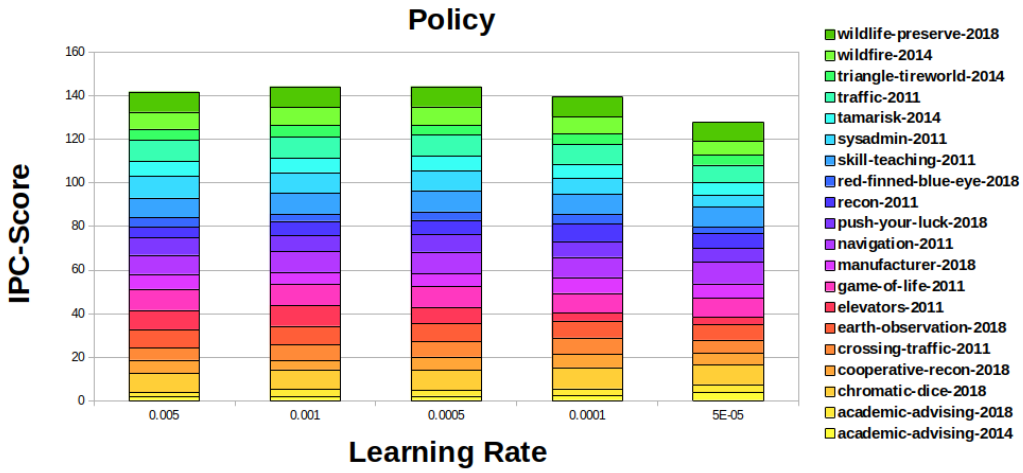

Figure 19: Results of the policy model parameter exploration - learning rate

5.4 Batch Size

The batch size indicates how many sample points are fed to the NN at once. This has a direct influence on the calculated loss, since it sums up the difference between prediction and target for every single data point in a batch set. The adjustment of the model weights is done after each batch run, where the update of the neurons is a non-negligible task, in terms of computational effort. Particularly large batches allow a significant time saving, when training NN's. Since the Q-value NN generates a lot of training data, depending on the amount of applicable actions, possible outcomes and number of training states, one could assume that the larger the batch size is chosen, the better results can be expected. As the following results show, the opposite is the case. With increasing batch size, ipc-score decreases uniformly. The reason for the declining performance is that larger batch sizes take more time to converge and therefore also require an increased number of epochs. The diagram clearly shows that a batch size of 500 and 1000 results in almost the same ipc-score. In the following experiments the batch size of 1000 will be applied.

Figure 20: Results of the Q-Value model parameter exploration - batch size

Since a smaller batch size generates more batches per epoch, the number of neuron updates increases. Due to the increased amount of NN optimizations, more time is needed, which means a lower amount of generated NN's as the chart below highlights.

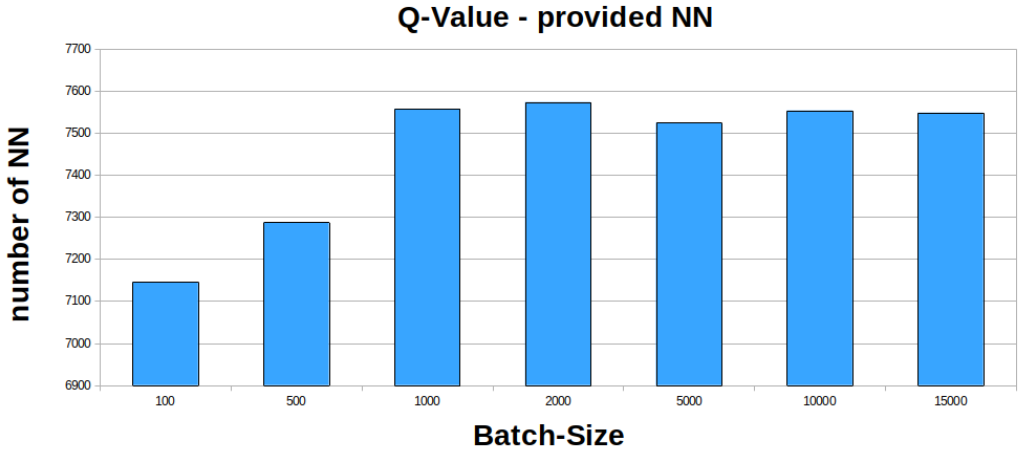

Figure 21: Amount of NN provided in Q-Value model generation procedure for increasing batch size

While the difference in the number of generated NN's is not huge, we assume that increasing the number of hidden layers would also increase the time consumption and thus amplify the effect.

In contrast to the Q-value NN, the policy variant shows a smaller decrease in the ipc-score. The policy NN uses a considerably smaller pool of training data, since it directly feeds the loaded training states to the model. Although there are scattered individual instances that provide up to 47000 states, most are limited to between 1000 and 5000 and even some are below 1000. If the total amount of training data is less than the selected batch size, they are all fed to the NN at once. It is noticeable that especially domains with a number of states in the upper 4-digit range are handled more poorly by higher batch sizes, such as *elevators* or *traffic*. In case of the policy NN there is a slight deterioration noticeable in the growing batch sizes as well. Overall the batch size of 2000 dominates the experiments, which is why this value is chosen for the following trials.

Figure 22: Results of the policy model parameter exploration - batch size

5.5 Hidden Layer

The depth of an NN, i.e. the number of hidden layers has a significant impact on training in terms of the required duration, as well as time needed to generate predictions. The reason for this is simply the fact that any signals have to travel through more neurons. On the one hand, just as with the number of epochs fewer NN are produced in the same time.

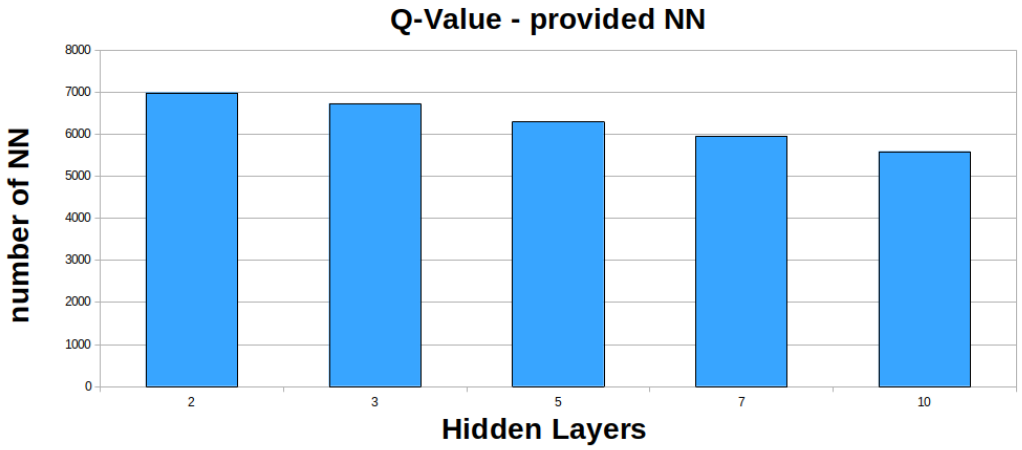

Figure 23: Amount of NN provided in Q-Value model generation procedure for increasing number of hidden layers

On the other hand, the predictions take longer, which significantly reduces the number of trials within a run. Fewer attempts have the consequence that the probability of better results decreases.

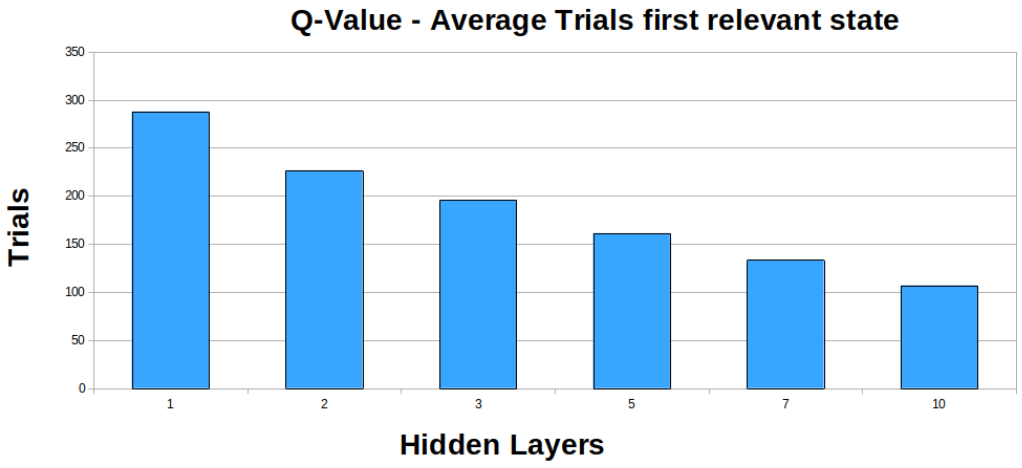

Figure 24: Number of trials first relevant state in Q-Value model for increasing number of hidden layers

A much more serious consequence is that if the prediction time is increased too much it can happen that the time for the single step is not sufficient. As a direct consequence,

there are some instances that could not achieve not a single result. This behavior occurred mainly with the larger NN's, in the *wildfire* and *academic-advising* domains. Knowing this, it is not surprising that the larger NN's perform significantly worse, which is also reflected in Figure 25.

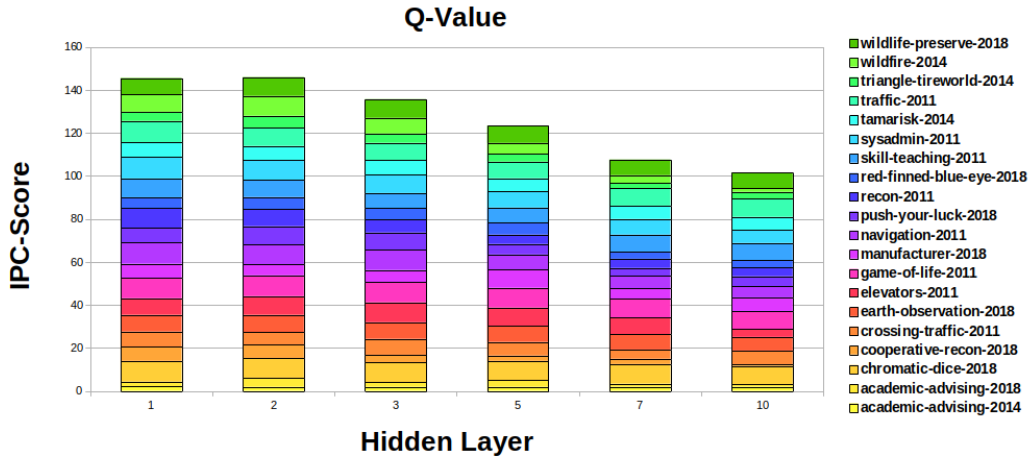

Figure 25: Results of the Q-Value model parameter exploration - hidden layer

In the case of policy NN, the number of trials also reduces with increasing depth. In contrast to the Q-value variant, however, considerably more trials are performed here at all.

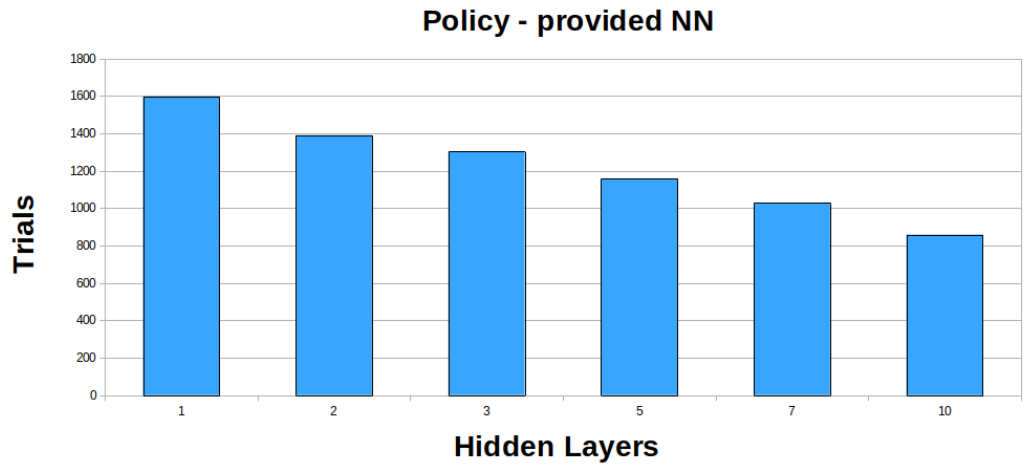

Figure 26: Number of trials first relevant state in Q-Value model for increasing number of hidden layers

Finally, only one prediction needs to be calculated for the evaluation of the actions, which generates a corresponding value for each applicable action. In contrast, the Q-value NN must generate a prediction for each state-action pair, which means a lot more computational effort if there are a correspondingly large number of applicable actions. The experiments with a larger layer do not offer as bad a performance as those of the previous overview, anyway a better result is also shown here in the number of hidden layers 1 and 2. Now that this parameter has also been determined, we have all the necessary components for the setup in the direct comparison with PROST2014. The sequential optimization approach applied in this chapter provides a local optimum. In this way, we aim to enable an assessment of the potential of the policy and Q-value NN procedure.

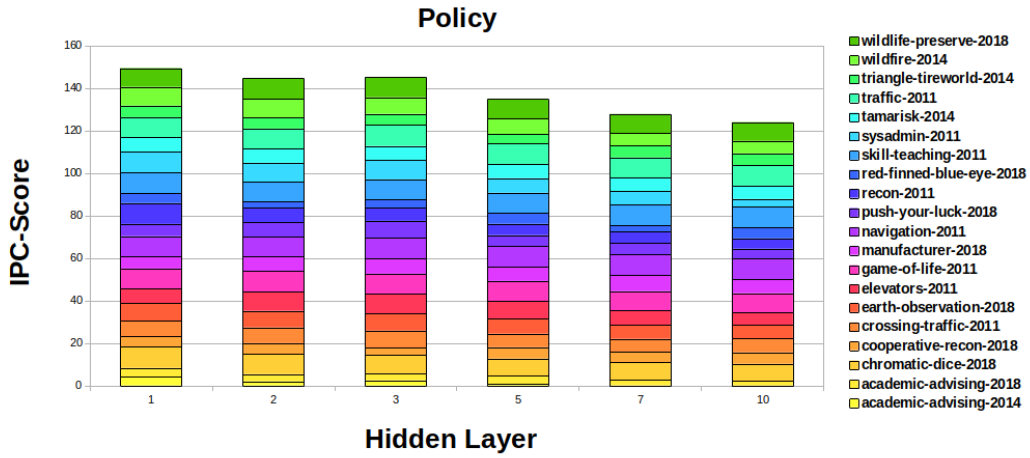

Figure 27: Results of the policy model parameter exploration - hidden layer

6 Evaluation and Adjustments

In this section we discuss the results obtained from the previously determined parameters. PROST2014 serves as a reasonable comparison, as it is state of the art for planning tasks in MDP environments. In addition, we look at further optimization possibilities with regard to the functionality and ingredients of the NN.

6.1 Results

As mentioned in the last section, we review the two best results for each NN variant, whose results are shown in the following graph. To ensure that the final evaluation did not produce the results through an unlikely sequence of coincidences, we executed the final experiment in triplicate. In doing so, all three evaluations were in similar value range like the one presented here and had no noteworthy differences from the other reports, which indicates small confidence intervals.

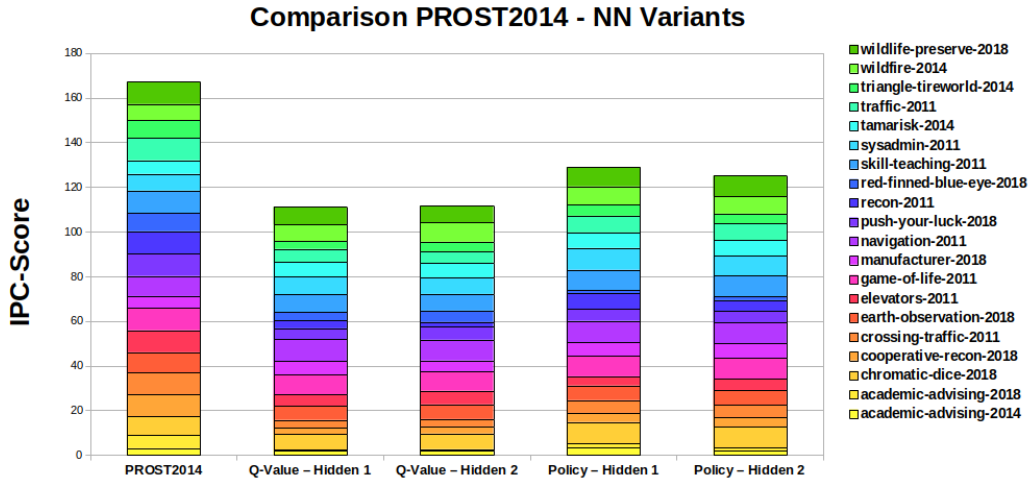

Figure 28: Final results including PROST2014, Q-Value NN and policy NN

It can be clearly observed that PROST2014 is dominating the overall ranking. Nearly all domains are processed significantly better than the NN procedures do. Nevertheless, it can be recognized that the NN's are quite capable of learning a decent performance, otherwise they would not be able to achieve similarly good results, at least in some cases. For example, the policy NN's shows remarkable progress in the *chromatic-dice* field, but also the *sysadmin* and *tamarisk* instances are performed better than it is the case in PROST2014. A likely reason for this could be the structure of the domains themselves. After all, these have a very

wide occurrence of successor states. Due to the high spreading, it might be rather difficult for PROST2014 to make a reasonable overall decision that takes into account all successors and the large spectrum of applicable actions, than in other domains. With a suitable data basis in the form of training states, this seems to be easier for the policy NN to realize. On the other hand, there are also domains like *red-finned-blue-eye*, which turn out considerably worse compared to the dominant planner. This could be due to the fact that the number of training states is comparatively low. Despite the limitation of the number of sampled possible outcomes using the introduced *threshold* variable, the generation of the NN for a too large set of states was not possible due to precense of too many actions. In particular, the latter instances of *red-finned-blue-eye* 6, 8 and 10 have a large amount of actions, with 681, 817 and even 2368 number of actions. While there are other domains with very many actions as well, most of them are often not applicable at the same time, which greatly reduces the amount of outcomes. Even the Q-value variant shows its strengths in the *wildfire* domain. The setup with hidden layer 2 is particularly striking, since it produces significantly better results than all other heuristics. Despite the rather poor overall result, there seems to be some potential in this approach as well. One reason for this outcome could be that the training states, generated by random walks, were of comparatively good quality and thus provided more important information to the NN. Nevertheless, it is recognizable that in general the policy variant outperforms the Q-value NN's. It is plausible that this is related to the number of trials shown in Figures 24 and 26 that an NN is capable of performing within the time constraint. Thus, the policy NN is able to achieve a value more than five times higher than the Q-value NN.

6.2 MSE Loss Function

In addition to the previously used MSE loss, we introduced the MAE in Chapter 3. By quadratically calculating the difference between target and predicted value, small deviations are attenuated and larger ones are amplified. This has the effect that the model converges faster, but also brings a disadvantage. The MSE is extremely susceptible to single outliers, since this single error is weighted very highly by the squaring of the difference. In the case of the domains used here, we must definitely expect larger outliers. Nevertheless, the experiments were also carried out with this loss function, but only with moderate success. Many of the instances failed and therefore did not provide a comparable outcome, since only 122 of 200 instances provide a result. Such a misbehaviour occurs when the loss function grows beyond some border, because a too high value interferes with the updating

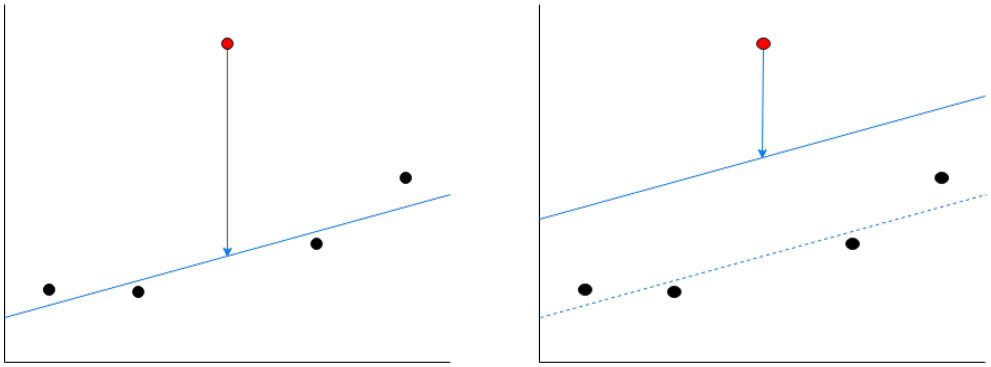

Figure 29: Linear Regression adaption using MSE on set of sample points including outlier

of the neurons and subsequently gives erroneous predictions. Murphy (2012) confirms this shortcoming for training sets including such outliers as in our environment. One proposed solution is to use the MAE, as it promises a more robust approach. In addition to the problematic nature of the loss, caused by isolated outliers, a consequential error must also be considered. As shown in Figure 29 the outlier shifts the prediction function to reduce its own loss, but this increases the loss for all other sample points. As the respective loss is squared, the loss of the outlier has a greater influence on the updating of the neurons. It is therefore not surprising that even the runs that worked in the experiment produced similar but sometimes worse results than those with the MAE.

6.3 Bounded Networks

In the processes so far, we have alwawys tried to generate as many NN's as posible in the respective domains. The previous assumption was that for deep the corresponding NN represents the best approximation of the actual Q-value. The question arises, however, whether an NN of a smaller depth might make equally good or even better predictions. For this reason, we introduce a new variant of NN's, the *bounded network*. Actually this is the same kind of NN that was produced before by the Q-value or policy approach, with the adaption that the model of the respective depth d, i.e. NN_d, is no longer used in this depth. Instead, a boundary value is chosen in advance and defines up to which depth NN's are used. If a boundary of 8 is given, this NN is used for all depths from 8 to the horizon and the smaller ones as before also in descending order according to their numbering. The intention is to examine the process for a generally suitable depth that is already close to the optimum. Since there are several instances that have already produced only a few NN's in the generating process, it is most reasonable to search only in the low value range for a

suitable boundary. In the following graph we show such an experiment and the performance of the used Q-value NN, restricted by different boundaries, compared to the PROST2014 and the normal procedure of the experiment shown in the previous section. It turns out that a lower number of NN used leads to a uniformly decreasing final result. However, it should be mentioned that already at a depth of 12 there is only a slight difference to the normal variant with the use of all provided NN's.

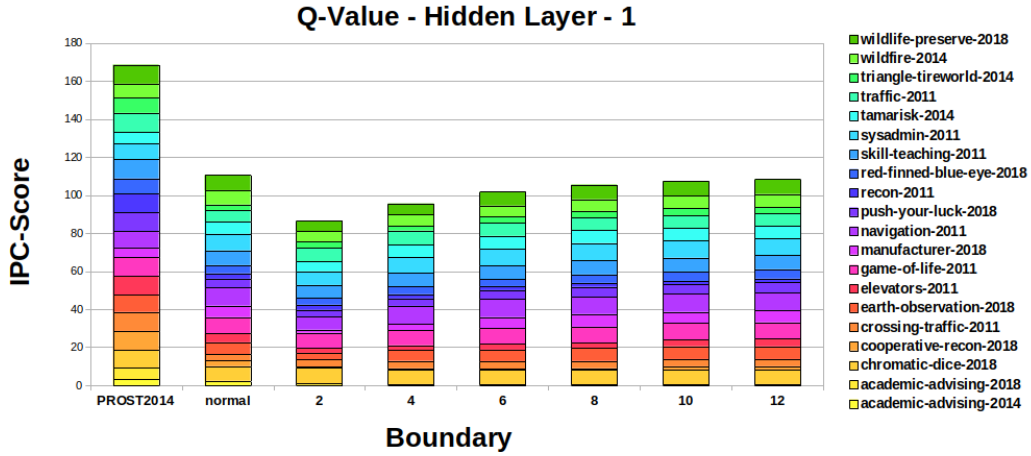

Figure 30: Results of the bounded Q-Value model

In addition, while the bounded NN's perform poorly overall, there are individual instances where the limitation not only performs better than the normal variant, but even overcomes the PROST2014. These are mainly the easier, i.e. the smaller tasks of the respective domains. One reason for this could be that in the smaller state-spaces, it may be possible to reach an optimum early on. For example, an optimum in the *elevators* domain would be reached when all passengers are at their destination floor and the elevators only have to wait. Once such a state has been reached, it can only be negatively influenced by its successors. Since there are no passengers to move, it is not possible to perform target oriented actions, but it is always possible that new passenger appear depending on chance, which again leads to a deterioration of the reward. In this way it can be that a model of a certain depth, already knows the best path due to its training samples, but its successor unlearns this knowledge again. This explains why, for example, in the first *elevators* instance, the bounded variant limited to depth 6 performs best. The situation is similar for the *game-of-life* domain, where it can be recognized that the first five instances overcome using a setup with a boundary of 10 the normal Q-value setup, as depicted in the diagram

below. The performance drops off quickly for higher instances, since these tasks are usually much more complex, in terms of the amount of state variables and actions that have to be observed, there is probably no local optimum that can already be determined at a low depth, i.e. with a low number of applied actions.

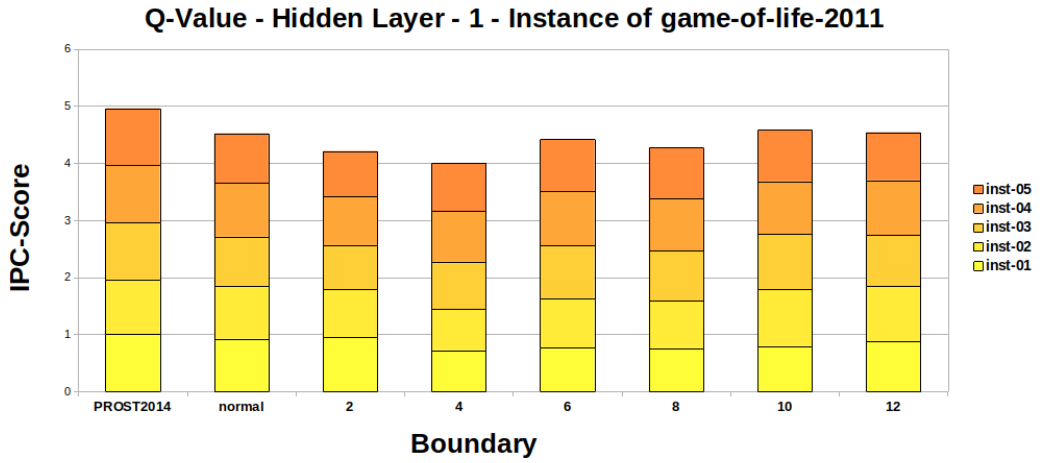

Figure 31: Results of the bounded Q-Value model in game-of-life-2011 domain, instances 1 to 5

The Figure 32 illustrates the bounded trial of the policy NN. The behavior is similar to the previous experiment, with even fewer instances reporting any progress.

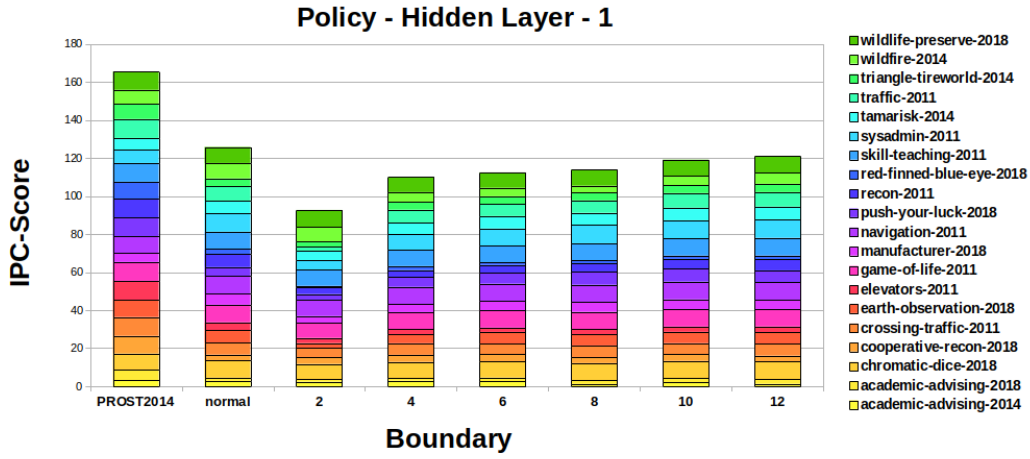

Figure 32: Results of the bounded policy model

However, it turns out that in some cases, a lower number of NN at least achieves a comparable result for certain domains. If further tests confirm this assumption, the time required to generate the NN could be significantly reduced if a model from a lower level does not bring any improvement anyway.

6.4 Entire Set of Training States

In Chapter 3 we introduced a breadth first search algorithm to create the total amount of states in a certain MDP state-space. In this section, we demonstrate another experiment that executed the planning search based on this training set. As we have already noted it is only possible to generate all states in smaller domains as this is intractable for large MDP's. For this reason, there are only 53 out of 200 results and a general comparison is not possible and we restrict ourselves to comparing a few instances.

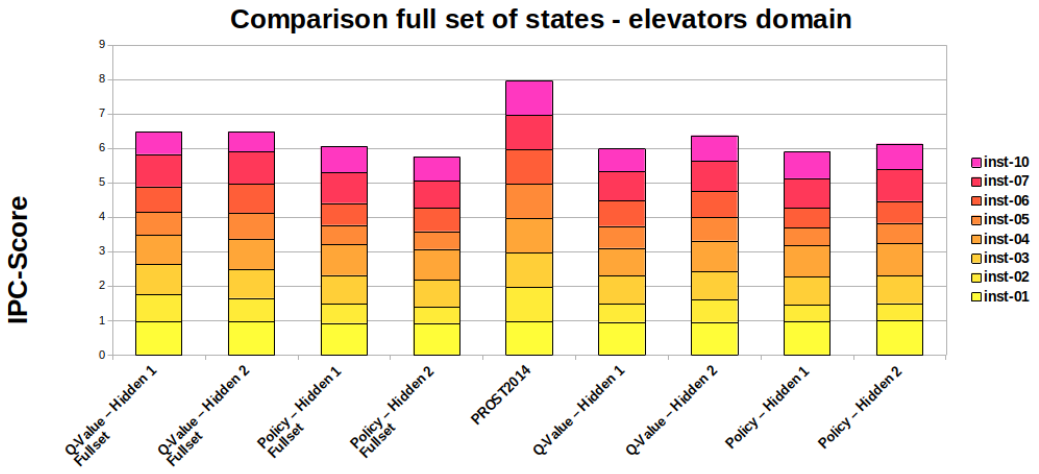

Figure 33: Results of the models generated by entire set of states

For comparison, we use the *elevators* domain, since it was in this category possible to create NN's for most of the instances. In the graph we find in the middle PROST2014 and on its left the results for the training sets with the entire collected state-space. Despite the increased amount of data, we were not able to outperform PROST2014 in the overall rankings. At least, few tasks like instance 2 of policy NN with 1 hidden layer were able to prevail over the previous trials with restricted training set. Nevertheless, one has to admit that the expected improvement did not arrive. One possible reason for this could

be that the most important states in the *elevators* domain could already be determined with the help of performed random walk simulations, so that the additional emergence of further states little added value in terms of information gain. Also in other domains no considerable improvements occurred, whereby however also the quantity of instances that are analyzable is rather small due to the intractable computational effort when generating the entire states of the state-spaces. A general statement about this procedure has to be elaborated by further tests.

7 Conclusion

To determine the applicability of NN's to learn Q-values in different MDP environments, several experiments were performed and evaluated. In this process, we were able to determine that of the two variants introduced, the policy NN outperformed the other one. Anyway, it turned out that the PROST2014 could not be surpassed in the overall results and this despite the separation of NN generation and actual search, which gives these variants a significant time advantage. On the other hand, there were quite promising approaches that delivered an acceptable result at least in some domains. In this way, several domains such as *chromatic-dice*, *manufacturer*, *navigation* and *sysadmin* exceeded the outcomes of PROST2014. This result is satisfactory as a first attempt and indicates that by further refinement of the components applied, a solid heuristic can be provided.

One of the core problems, if not the major one, is the quality of the training states. Without suitable states, i.e. with sufficient and meaningful information content, the NN's cannot learn relationships that are relevant in the respective domain for decision making. Although, we demonstrated a opportunity to compute more, or even all states of a state-space, this is exactly the procedure that the process of planning is supposed to prevent. Furthermore, it was shown that even a brute force method does not necessarily lead to an optimal, or at least better result than the PROST2014. The quality of states is domain specific and must be assessed in advance. In the case of the *sysadmin* domain, states where all computers are already turned off extend the knowledge only slightly. A reasonable assumption would be to use only states that have a minimum number of active computers.

Besides the provision of data, there is still room for improvement in the ingredients of the NN itself. As already mentioned, we have so far only performed a local parameter search, which reflects a trend but does not lead to a globally optimal setting. One reason for this is the fact that the individual parameters exert an influence on each other. So if, as in this paper happened, a certain number of epochs has been specified on the basis of a first experiment, only a certain range of the learning rate can be calibrated to this parameter. Because if the number of iterations is not sufficient, the step size will be too small and the target function cannot be reached at all. The same holds for the identification of the amount of hidden layers postprocessing and vice versa. For this reason, the best values can be obtained by searching in a wide range of values and comparing several constellations. Furthermore, it could be promising to determine a domain specific dynamic adjustment of the parameters, as it is the case with the NN breadth, which is equal to the input layer size.

The same applies to the variant of bounded NN's since we assume that there is an

individual optimum for each instance. A reasonable alternative could be another modified NN variant, which feeds, in addition to the state variables, the current depth to the model. Such a *horizon NN* would have the advantage of combining the NN's that were previously generated individually all into one. By using a single NN for the entire horizon, the training can be easily distributed into multiple learning units and the data is generated by the NN itself by feeding the corresponding depth.

The parameter alignment of Chapter 5 was performed based on the SGD optimizer, however there are other optimizers which have further setting variables to enable an even more precise adaption to the respective task. Furthermore, in many applications of NN, the MSE loss is used to speed up the training procedure. In our setup, the use of this loss function was not possible due to occurring outliers. Nevertheless, the MSE loss could be applied by normalizing the values received of the reward function to a smaller range of values.

A commonly used approach is reinforcement learning to allow the model to explore the MDP environment on its own and learn reasonable actions by doing so. Since such an agent is initialized without prior knowledge, the agent would have to start acquiring a knowledge base by performing random walks. To speed up this laborious process, it would be an interesting approach to use the supervised learned NN's, introduced in this thesis, to provide a rough direction of the state-space in which reasonable states are more likely to lie.

In conclusion, the experiments presented here achieved comprehensible results and also reveal promising possibilities for extensions. The large number of opportunities to conduct further research in this area demonstrate that the topic of this thesis still offers interesting branches of research.

Bibliography

[1] Bahar, Iris R.; Frohm, Erica A.; Gaona, Charles M.; Hachtel, Gary D.; Macii, Enrico; Pardo, Abelardo; and Somenzi, Fabio (1997). *Algebraic decision diagrams and their applications.* In Proceedings of the International Conference on Computer Aided Design (ICCAD 1993), 171–206.

[2] Bellman, Richard (1957). *Dynamic Programming.* Princeton University Press

[3] Boutilier, Craig; Dearden, Richard; and Goldszmidt, Moisés (2000). *Stochastic Dynamic Programming with Factored Representations.* Artificial Intelligence (AIJ) 121(1–2), 49–107.

[4] Browne, Cameron; Powley, Edward J.; Whitehouse, Daniel; Lucas, Simon M.; Cowling, Peter I.; Rohlfshagen, Philipp; Tavener, Stephen; Perez, Diego; Samothrakis, Spyridon; and Colton, Simon (2012). *A Survey of Monte Carlo Tree Search Methods.* IEEE Transactions Computational Intelligence and AI in Games 4(1), 1–43.

[5] Geißer, Florian and Speck, David (2018). *PROST-DD - Utilizing Symbolic Classical Planning in THTS.* IPPC 2018

[6] Haykin, Simon (2009). *Neural Networks and Learning Machines.* Prentice Hall

[7] Keller, Thomas and Eyerich, Patrick (2012). *PROST: Probabilistic Planning Based on UCT.* In Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012), 119–127.

[8] Keller, Thomas and Helmert, Malte(2013). *Trial-based Heuristic Tree Search for Finite Horizon MDPs.* In Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013), 135–143.

[9] Kleene , Stephen Cole (1952). *Introduction to Metamathematics.* Princeton

[10] Kocsis, Levente and Szepesvári, Csaba (2006). *Bandit Based Monte- Carlo Planning.* In Proceedings of the seventeenth European Conference on Machine Learning (ECML), 282–293.

[11] Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David; Rusu, Andrei A.; Veness, Joel; Bellemare, Marc G.; Graves, Alex; Riedmiller, Martin; Fidjeland, Andreas K.; Ostrovski, Georg; Petersen, Stig; Beattie, Charles; Sadik, Amir; Antonoglou, Ioannis;

King, Helen; Kumaran, Dharshan; Wierstra, Daan; Legg, Shane; and Hassabis, Demis (2015).*Human-level control through deep reinforcement learning.* Nature 518, 529–533

[12] Murphy, Kevin P. (2012). *Machine Learning - A Probabilistic Perspective.* MIT Press

[13] Puterman, Martin L. (1990). *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley

[14] Russel, Stuart J. and Norvig, Peter (2010). *Artificial Intelligence - A Modern Approach.* Prentice Hall

[15] Sanner, Scott (2010) *Relational Dynamic Influence Diagram Language (RDDL): Language Description.*

[16] Stathakis, Demetris (2009). *How many hidden layers and nodes?.* International Journal of Remote Sensing, 2133-2147

[17] Sutton, Richard S. and Barto, Andrew G. (1998). *Reinforcement Learning: An Introduction.* MIT Press